Chapter 7

Introduction to the CNN Universal Machine

We have seen in Chapter 6 that not all tasks can be implemented by a single CNN template, the XOR function is a typical example.

There are many tasks which are solved by applying several templates, or by applying one template several times. If we consider a *template as an instruction* with well defined input and output, we can define a *CNN subroutine* or *function* (as in C like languages)when applying several templates. We can build up processes and complete programs from functions and other instructions.

We define a subroutine by specifying the following items:

- the input/output parameters,
- the global task,
- the informal description of the algorithm,
- the CNN implementation.

In this chapter the CNN implementation is given by 3 equivalent ways:

--the *hardware schematics*, supposing each CNN template (placed in the CNN Software Library) is implemented by a separate device containing discrete hard wired cells and additional local (cell by cell) and global devices.

- a *flow diagram of the CNN algorithm*, and

- a list of *consecutive instructions*, henceforth called a *program* written in a simple vocabulary involving the CNN *analog* and *logic* operations, henceforth called an analogic CNN *language*, or simply "α" *language*.

An *Alpha Compiler* is supposed to exist to translate the code into executable programs un CNN chips. We will describe this process later in Chapter 9.

Indeed, we follow the theory and practice of digital computers. According to the classic Turing-Church thesis, each algorithm defined on integers or on a finite set of symbols (e.g., "yes' or "no") can be equivalently expressed by

- a Turing *machine*,
- a recursive function (an *algorithmic description* using a finite set of elementary operators), and
- a program defined on a computer using a *language*.

As to the α *language,* the key instruction is the CNN template operation defined as

TemplateName(InputImage,InitialStateImage,OutputImage,TimeInterval,BoundaryCond)

for example

```
EDGE (LLM1, LLM2, LLM3, 10, -1)
```

means that an edge detector template called EDGE is applied with input, initial state, and output images denoted/stored by/in LLM1, LLM2, LLM3 images, the output is taken at time t=10 (measured in the time constant of the CNN cell, τ_{NN}), and the fixed boundary value is -1.

7.1 Global clock and global wire

Definition: a component is called *global* if its output depends on all cells of the array or its output effects all cells of the array.

Like in any programmable system we need a clock. To emphasize that within one CNN template operation there is no clock, we will call our clock as a *global clock* (GCL). This means that during one clock cycle entire array of cells implements the same template instruction.

The global clock is used to control a set of switches (enabling, disabling, latching functions) which provide that at a given clock cycle only the prescribed signal route is open.

In many cases we have to decide whether any black pixel remains in the processed image, i.e. whether it is completely white or not. We call the operation GW(.) which tests this property (it is called “global white”, “global wire”, or “global line” in the literature).

GW(.) is defined as follows:

Given a binary image P containing MxN pixels.

$$\text{GW(P)}=\begin{cases} 1(\text{Yes}) \text{ if all the pixels of P are white} (-1) \\ -1(\text{No}) \text{ if at least one pixel in P is black} (1) \end{cases}$$

in some implementations “NO” is represented by 0.

7.2 Set inclusion

We want to detect whether

$$S_1 \subset S_2$$

S_1 and S_2 are represented by pictures P_1 and P_2, respectively. A pixel is black if the corresponding element is included in the given set. $P_1 \subset P_2$ if and only if all black pixels defining P_1 are elements of the black pixels representing P_2.

Now we following define the subroutine or function SUBSET 1(., ., .).

SUBSET 1(P1, P2, Y)

P1, P2: binary images of size MxN, the black pixels are representing the relevant sets.

Y: logical value, Yes or No, represented by 1 and -1 , respectively.

Global task: determine whether a set S2 defined on an Euclidean plane is a subset of another set S1.

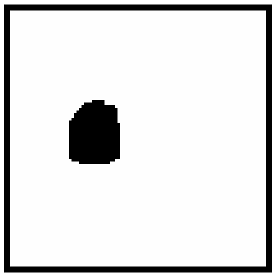
Algorithm

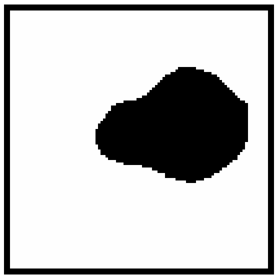
Given P_1 and P_2. The algorithm consists of 3 steps:

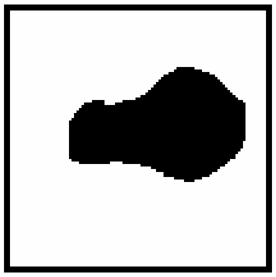
```
P3:= NOT(P1)
P1:= P3 AND P2
IF P1 contains only white pixels THEN
    Y:= 1 (Yes)    ELSE Y:=-1 (No)
```

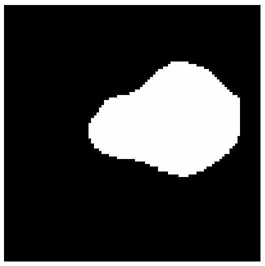
Remark: The NOT and AND operations are acting pixel by pixel.

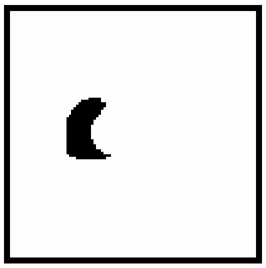
Example 1:


set P2

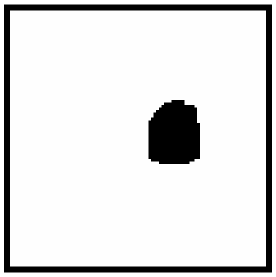

set P1

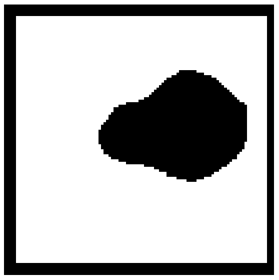

P1 ∪ P2

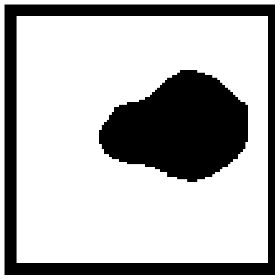

NOT P1

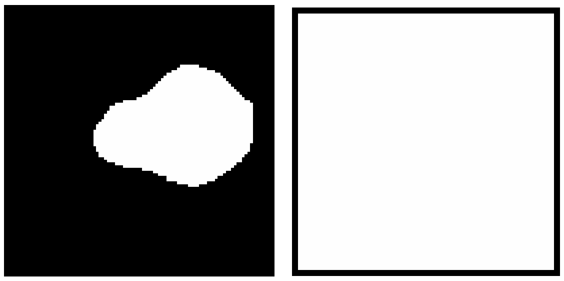

(NOT P1) AND P2

Example 2:


set P2


set P1


P1 ∪ P2 = P1



NOT P1 (NOT P1) AND P2

CNN implementation

A: Hardwired components

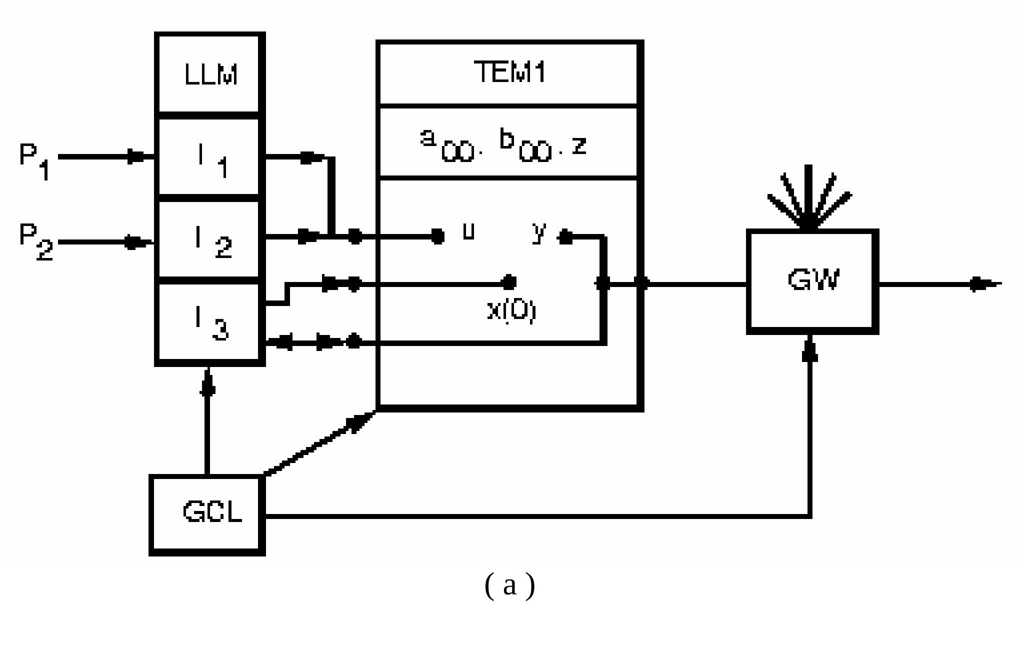
To implement this algorithm via CNN we need some additional components in each cells and 2 global components Suppose that we hard-wire the components. An extended cell (type 1) is shown in Fig.1(a), containing a (local) logic memory LLM with 3 storage places, the GW, and a clock. The latter two are global elements operating on the whole cell array.

We suppose that we have 2 different CNN arrays (cells and interconnections), one for implementing the NOT operation (LOGNOT CNN) and one for the AND operation (LOGAND CNN). These templates are shown in Chapter 3. The hard-wired solutions are shown in Fig.1(b).

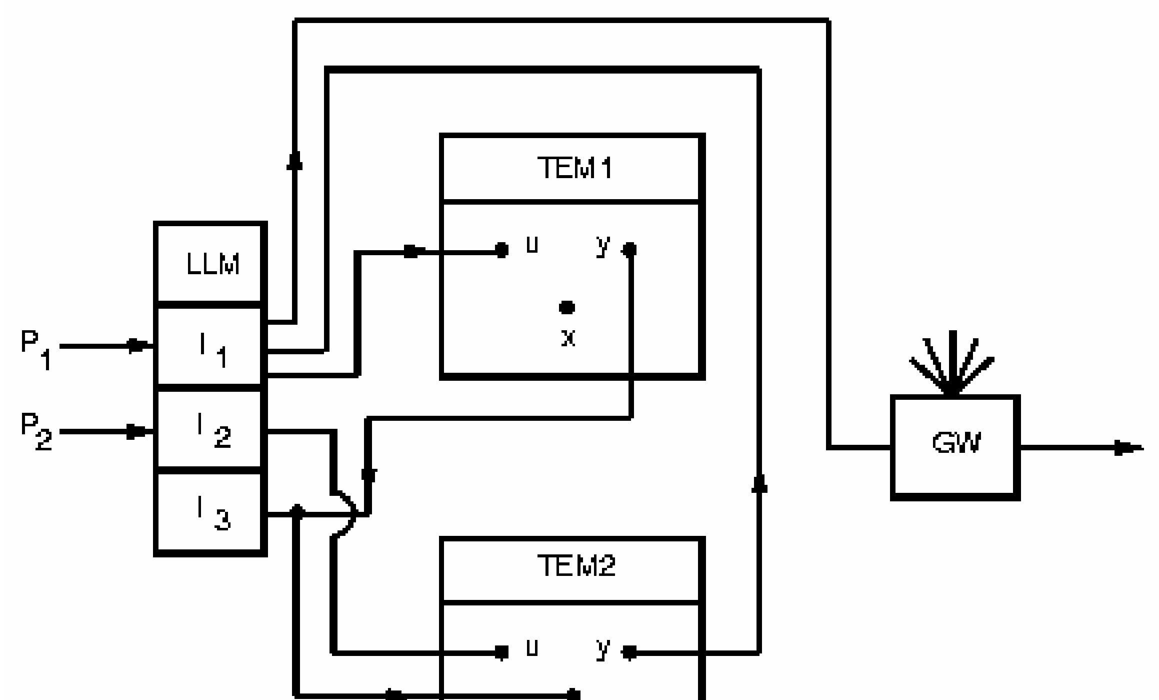
B: Flow diagram and program

If we place the extended cell in an MxN array, the following flow diagram will implement the function SUBSET 1(., ., .). In addition, in Fig.2, we show the α program as well.

Like in a digital programming language, our α language is using a few elementary instructions. Here, in addition to the template activation instruction, we use an instruction for the GW( ) test and memory copying instructions. We declare the templates to be used in the function by listing them between the brackets of the USE declaration.

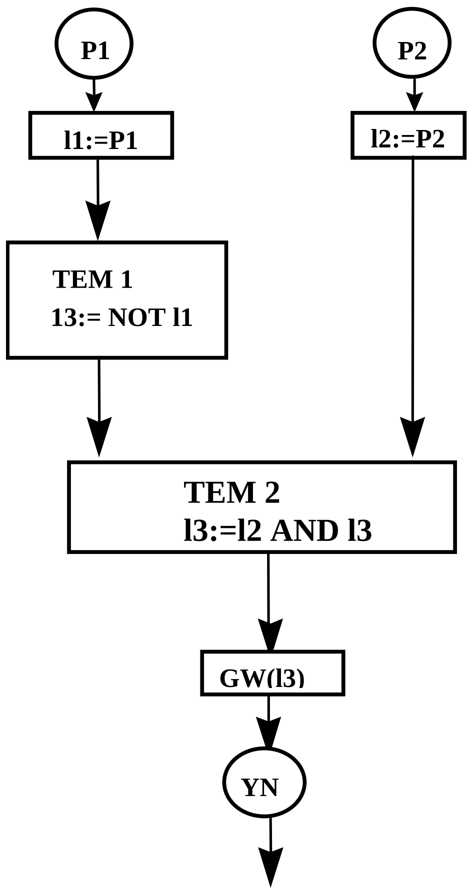


(a)



(b)

Fig. 1 (a) The extended cell 1. In addition to the CNN cell we have three new components: a local logic memory (LLM), a global white tester (GW) and a global clock (GCL); (b) the hard wired solution for SUBSET 1 (.).



```
FUNCTION  SUBSET 1;

USE(LOGNOT,LOGAND);

LLM1:= P1;
LLM2:= P2;

LOGNOT(LLM1,LLM1,LLM3,5,-1);

LOGAND(LLM2,LLM3,LLM1, 5,-1);

 Y:=GW(LLM1);

 ENDFUNCT;
```

Figure 2 The SUBSET subroutine as a function. TEM1 is LOGNOT, TEM2 is LOGAND

7.3 Translation of sets and binary images

We want to translate two-dimensional sets and binary images by a prescribed vector. This vector is given by its horizontal and vertical coordinates, m and n, respectively. The set S is represented by the black pixels of an image P. The translated image PT is given by its black pixels as well. Subroutine TRANSLATE(., ., ., .) performs this task

$\boxed{\text{TRANSLATE (P, PT, m, n)}}$

P, PT: binary images of size MxN
m, n: integers

Global task:
Translate image P by vector (m,n)
(we suppose m, n >0 if not, simple modifications can be applied).

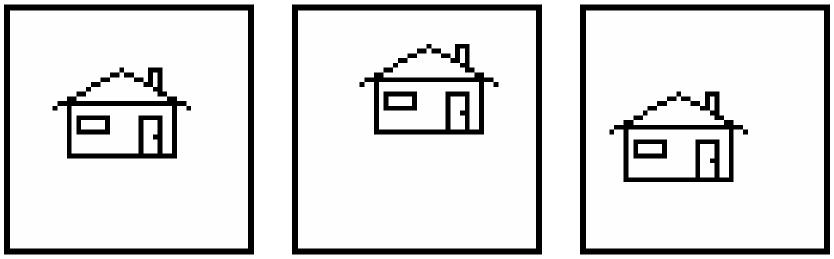
Algorithm

Given P, m, and n. The algorithm is performed in an iteration (a program loop):

```
PT:=P
FOR i=1 STEP 1 TO m
    PT:=SHIFT(PT, EAST)
FOR j=1 STEP 1 TO n
    PT:=SHIFT(PT, NORTH)
```

Here, SHIFT(PT, EAST) and SHIFT(PT, WEST) are the translating operators with one unit length to the EAST and NORTH directions, respectively.

Example:


Input SHIFT(4, 5) SHIFT(-4, -5)

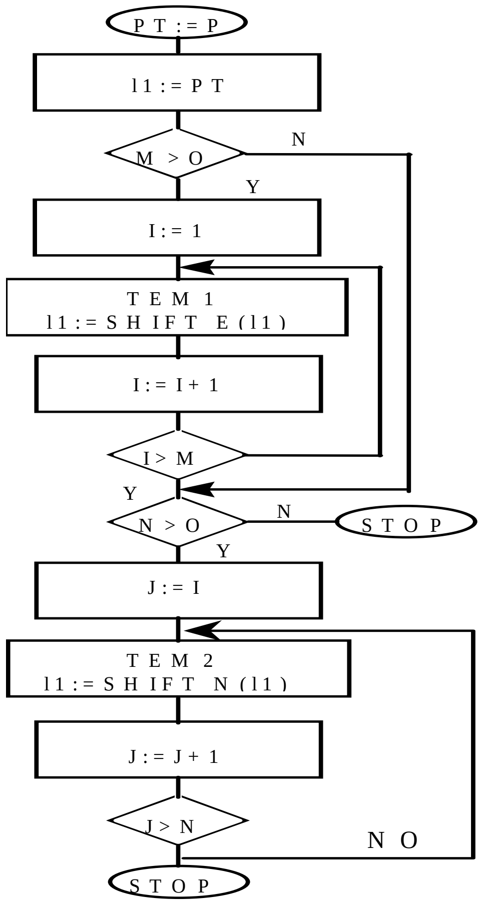
CNN Implementation

A: hardware components

For the implementation of this algorithm we do not need more components than we used in the preceding subroutine (SUBSET 1). The controlling mechanism, however, is more sophisticated. We have to check when to stop the iteration after m and n steps. This means we need a global control unit which controls the switches and stops/starts the iteration.

Again, we suppose that we have two different CNN cells (and arrays), one for the SHIFT to north, one for the SHIFT to east. However, now we need m and n samples of each CNN component, or we use these two components with a sophisticated control unit.

B: Flow diagram and program



```
FUNCTION TRANSLATE;
USE(SHIFTE,SHIFTN);

LLM1:=PT;

IF M>O THEN DO

REPEAT I:=1 TO M BY1

SHIFTE(LLM1,LLM1,LLM1,10,-1);

ENDREPEAT;

IF N>O THEN DO

REPEAT J:=1TO N BY 1

SHIFTN(LLM1,LLM1,LLM1,10,-1);

ENDREPEAT;

ENDFUNCT;
```

Fig 3.The function TRANSLATE

7.4 Opening and Closing and implementing any morphological operator

Two frequently used morphological operators are the opening and closing.
Opening is defined as: first erosion then dilation.
Closing is defined as: first dilation then erosion.
The difference is in the sequence of the two elementary templates.

We will show here the subroutine CLOSE(P, S, PC) where P is the original image, S is the structuring element, and PC is the result.

$\boxed{\text{CLOSE (P, B, PC)}}$

P, PC: Binary images of size MxN
S: 3x3 structuring element represented in a B template for erosion
the 3x3 feedforward template B defined by the structuring element with 1(black) and 0(white)for dilation, reflect B(centrally) to get B1 as the feedforward template

Global task

Given P, first apply a dilation, then an erosion with structuring element represented by B, defined above.

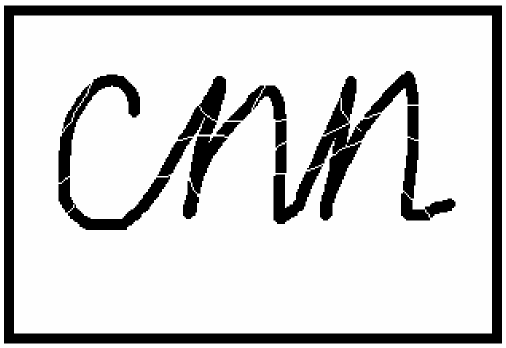
Algorithm

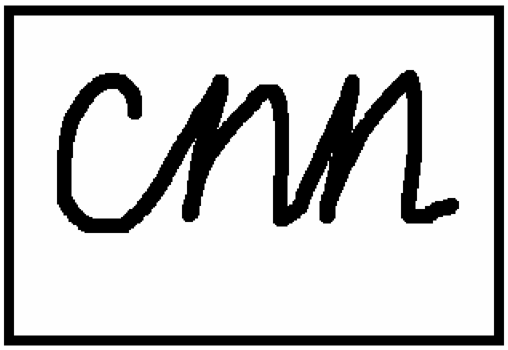
Given P and S (B). The algorithm has four steps:

```
P1:= P
P2:=DILATION(P1, B1)
P3:=EROSION(P2,B)
PC:=P3
```

Examples

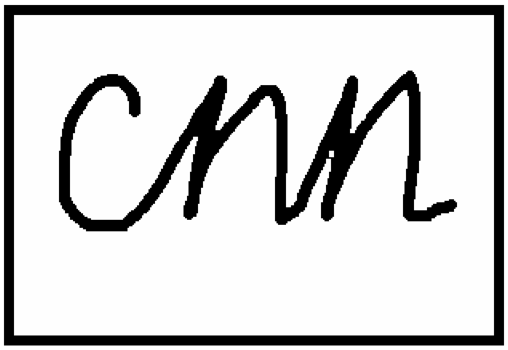
Example 1


input


output of DILATION operation

Example 2

input = output of DILATION operation


output of EROSION operation

CNN implementation

A: hardwired components

The hardwired schematics is very simple. Figure 4 shows it, we have two CNN components.

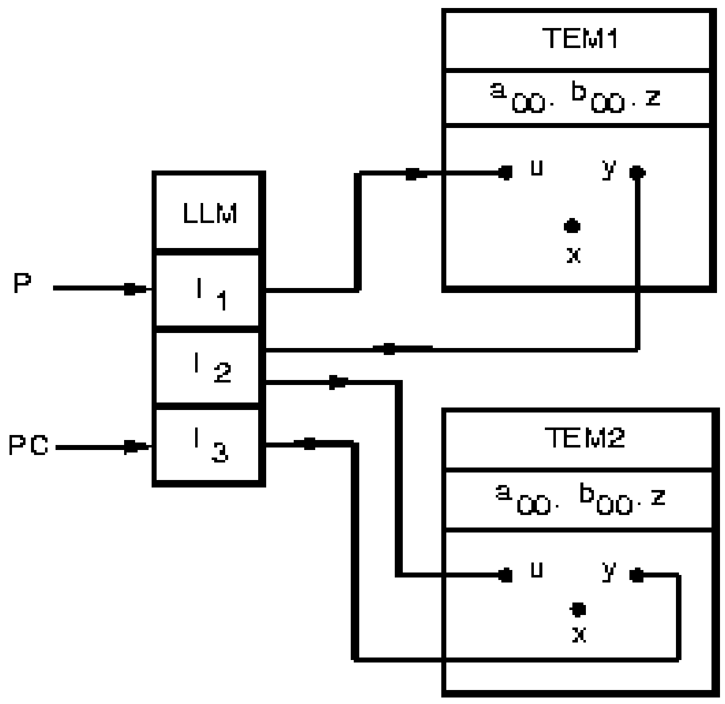
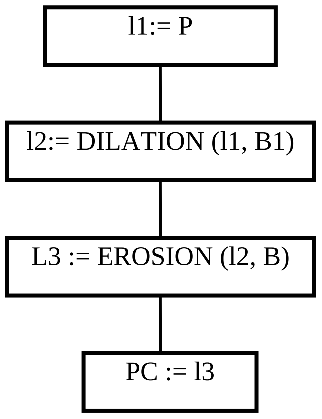


Figure 4

B: Flow diagram and program.



```
FUNCTION CLOSE;
USE(EROSIONB,DILATIONB1);
LLM1:=P;
xFill(0,ISTATE);

DILATIONB1(LLM1,ISTATE,
LLM2,10,-1);
EROSIONB(LLM2,ISTATE,LLM3,
10,-1);

PC::= LLM3;
ENDFUNCT;
```

Fig.5 The flow diagram and program of CLOSE.

Mathematical morphology has a calculus. Its deep mathematical foundations are well documented in textbooks[1] .

Opening of an image A by structuring element B is denoted by

$$A \circ B = (A ❍ B) \oplus B$$

where erosion is denoted by ❍ and dilation by $\oplus$, respectively.

Closing, denoted by $\bullet$ is, defined by

$$A \bullet B =(A \oplus B) ❍ B$$

opening and closing are dual operators.

$$A \bullet B = (A^c \circ B)^c$$

where ^{c} means complement. Hence, replacing A by A^c and complementing the result we get

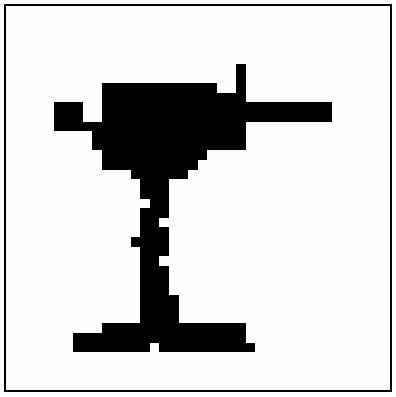
$$A \circ B = (A^c \bullet B)^c$$

We can implement this calculus by using a sequence of templates. The next Tables show an example. An image P is modified by a structuring element $\mathcal{S}$

[1] Daugherty, Introduction to mathematical morphology, SPIE, 1995

EROSION

$\mathcal{S}$ =


P

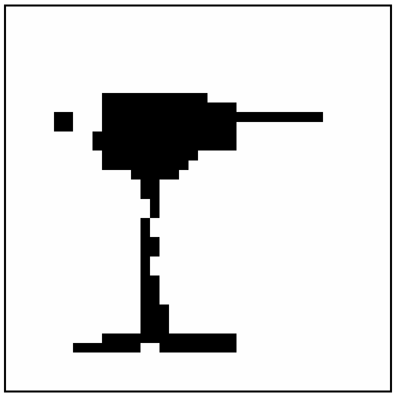
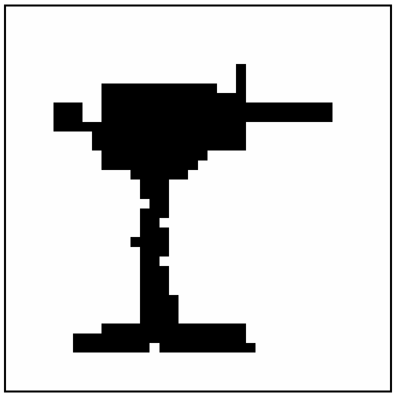

$P \ominus \mathcal{S}$

Table 1

DILATION

$\mathcal{S}$ =


P

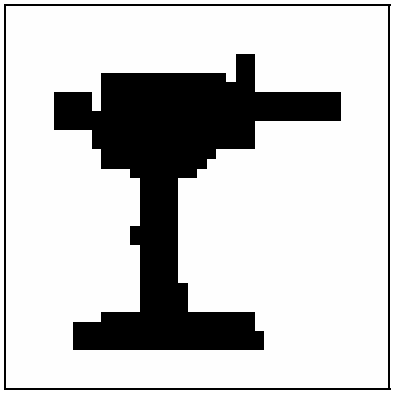
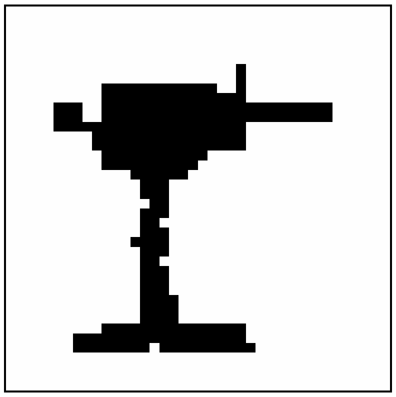

$P \oplus \mathcal{S}$

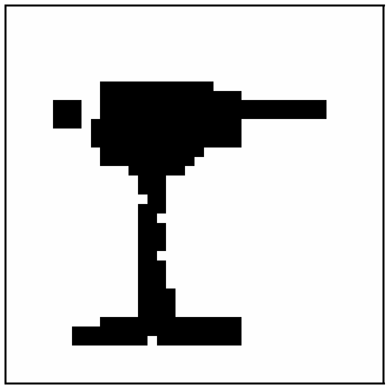
Table 2

OPEN

$\mathcal{S}$ =



P

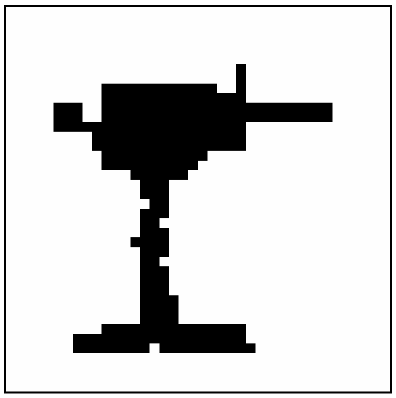


$P \circ \mathcal{S}$

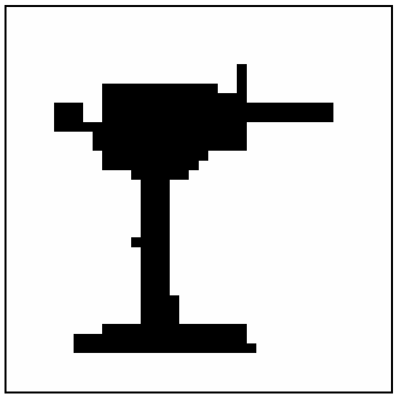
Table 3

CLOSE

$\mathcal{S}$ =



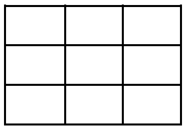
P

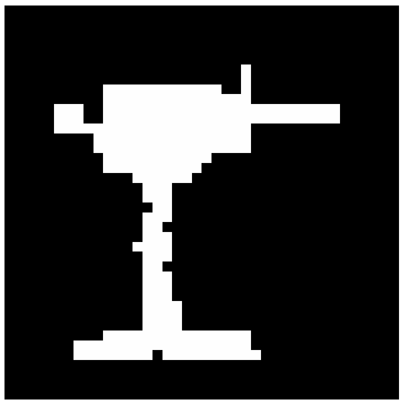
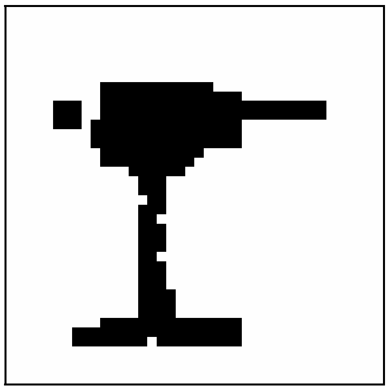


$P \bullet \mathcal{S}$

Table 4

OPEN

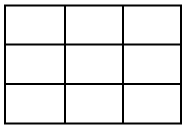
$\mathcal{S}$ =


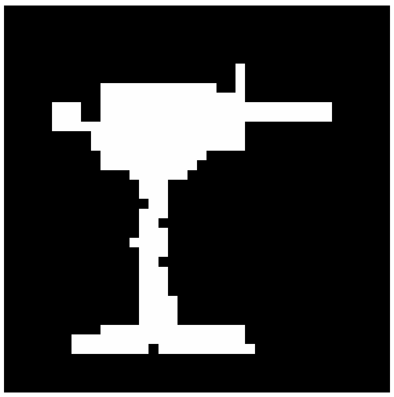
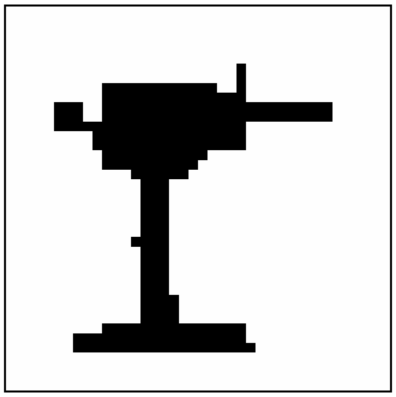



P^c $(P \bullet \mathcal{S})^c$

Table 5

CLOSE

$\mathcal{S}$ =


P^c $(P^c \circ \mathcal{S})^c$

Table 6

A *fundamental theorem of mathematical morphology*, the so called Matheron representation, asserts that a very large class of morphological operators can be decomposed into a union of erosions with a basis set of structuring elements. The art is to find the basis.

7.5. Implementing any prescribed Boolean transition function by not more than 256 templates

We have seen in section 6.6 that the XOR Boolean function cannot be realized by a simple CNN template, it is not linearly separable. On the other hand, we can realize it by applying several templates. The truth table is shown below:

	input		output
term	u_1	u_2	y
1	-1	-1	-1
2	1	1	-1
3	1	-1	1
4	-1	1	1

-1: false; 1: true

Using the minterm/maxterm notion, we can group the last two rows for generating the minterms by selecting those input combinations which yield outputs of logic 1:

$$m(u):\ u_1\bar{u}_2 + u_2\bar{u}_1$$

i.e., if one of the (now 2) minterms is true, the output $y=F(u_1, u_2)$ will be true. Similarly, for the terms with output of logic 0 , the maxterms (M(u)) are given by the first two rows:

$$M(u):\ \bar{u}_1\bar{u}_2\ .\ u_1u_2$$

i.e., if one of the maxterms is false, the output $y=F(u_1, u_2)$ will be false.

Hence, we can generate the XOR truth table by the sequential applications of two minterms, combining them with an OR function. Since the minterms contain AND and NOT functions, what we need, altogether, are the building blocks for AND, OR, and NOT functions. We have shown already the CNN templates for these 3 Boolean functions. Therefore, applying CNN operations, with different templates, iteratively, we can generate the XOR function.

There is a *systematic general procedure* for implementing any local Boolean function by the iterative application of different templates. For the CNN logic representation, we will use the convention: TRUE=1, FALSE =-1.

Having 9 inputs (u_1, u_2, ..., u_9) and one output we have $2^9 = 512$ output values (1,-1) for all the 512 input combinations. This means that we can generate *any* local Boolean function of 9 input/ 1 output variables, binary truth table by at moat 512 applications of different minterms, each one implemented by a CNN template. Next, we show a single, extended CNN cell which can be used to implement this procedure. Before, however, let us describe this procedure in an elementary flow diagram.

Suppose we want to calculate the output y_{ij} of 9 input Boolean function $Y=F(u_1, u_2, ..., u_9)$, u_1, u_2, ..., u_9 are the nine binary values of the cells in the neighborhood of cell C(ij).

F is given by the minterm, b_0, b_1,..., b_M ($M \leq 512$). In our single XOR example: M=2, b_0 and b_1 are the terms, $u_1\bar{u}_2$ and $\bar{u}_1 u_2$ respectively. These minterms can be coded as [1, -1] and [-1,1] and the procedure is shown in Fig.6.

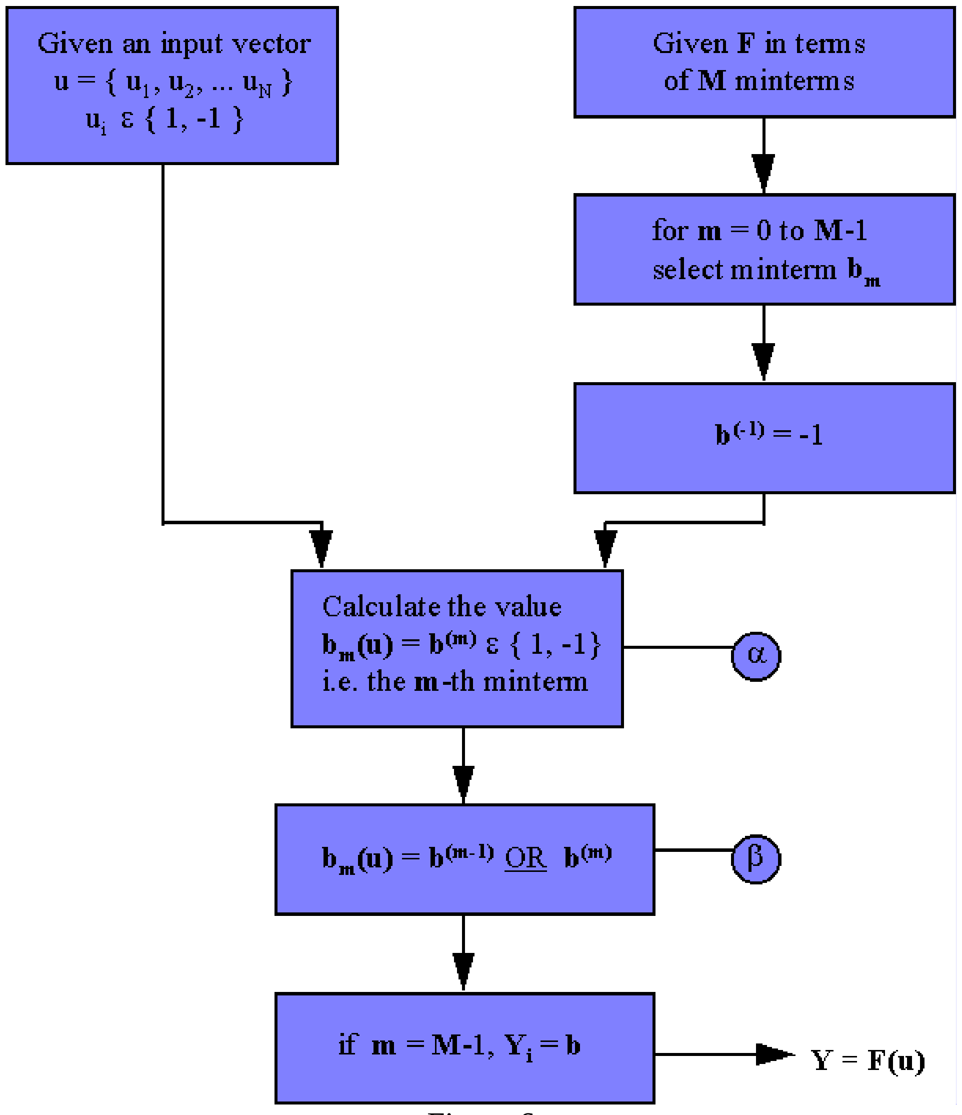

Given an input vector
u = { u1, u2, ... uN }
ui ε { 1, -1 }
Given F in terms of M minterms
for m = 0 to M-1 select minterm bm
b(-1) = -1
Calculate the value bm(u) = b(m) ε { 1, -1} i.e. the m-th minterm
α
bm(u) = b(m-1) OR b(m)
β
if m = M-1, Yi = b
Y = F(u)

Figure 6

In words, it means that we calculate the results of all the minterms at the given u (phase α) and make the ORing (phase β). To implement this flow diagram using our CNN templates we need the following building blocks.

- a logical storage for the given cell's input u_{ij},
- the CNN templates (A,B,z) for the minterms,
- an OR logic unit, and
- another 2-place logical storage (memory) with a shift (shift register)

This means that we need an extended CNN cell with the above units, in addition to the core CNN cell.

The extended CNN cell ij with its neighbors is shown below:

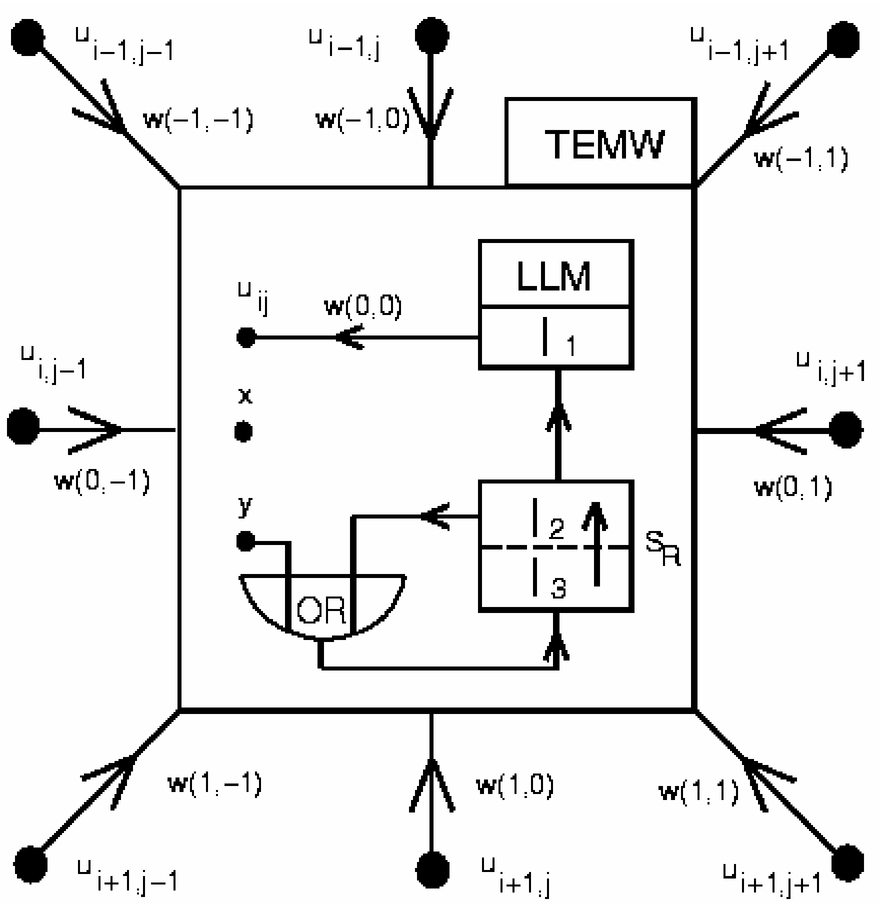


Figure 7

In the CNN cell we have an additional local logic unit (LLU): the OR gate. Suppose the 9 Boolean variables (u_1, u_2, ..., u_9) are placed on the inputs of all cells in the sphere of influence $S_r(ij)$. We have to find the template (A, B, z) for a given minterm then we can solve the problem. Next we will show this process.

A minterm is a linearly separable Boolean function. Referring to our earlier analysis, it can be shown (as an exercise) that the value of a minterm b_m at a given (u_1, u_2, ..., u_9) combination, $u_i \in \{-1, 1\}$ can be calculated by the following CNN template:

$$A = \begin{array}{|c|c|c|}\hline 0 & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 0 & 0 \\ \hline\end{array} \qquad B_m = \begin{array}{|c|c|c|}\hline w_{-1,-1} & w_{-1,0} & w_{-1,1} \\ \hline w_{0,-1} & w_{0,0} & w_{0,1} \\ \hline w_{1,-1} & w_{1,0} & w_{1,1} \\ \hline\end{array} \qquad z=-8$$

$$\begin{bmatrix} u_9 & u_8 & u_7 \\ u_6 & u_5 & u_4 \\ u_3 & u_2 & u_1 \end{bmatrix}$$

where the B template is coded using the minterm b_m in the following way. If in minterm b_m a variable is presented with its TRUE value, then the corresponding term $B_m(k,l)=1$, if it is FALSE $B_m(k,l)=-1$, if a variable does not exist $B_m(k,l)=0$.

This unit can be called restricted-weight threshold unit since the weights can take values from a finite limited set of values.

For example, minterm $u_1\, \bar{u}_2\, u_3\, \bar{u}_4\, \bar{u}_5$ is coded by a template

$$B_m=\begin{bmatrix} 0 & 0 & 0 \\ 0 & -1 & -1 \\ 1 & -1 & 1 \end{bmatrix}$$

Observe that one extended CNN cell can generate not only any minterm but, using the local logic unit and local logic memory, the final result of a given Boolean function of 9 variables as well. It is supposed that all input variables have their buffers.

If the number of zero outputs in the 9-input one-output Boolean truth table are less than 256, then there are less maxterms than minterms. Hence it is practical to code the maxterms. This can be done using the same extended CNN cell except the local cell logic contains a NOT and an AND gate.

There are more efficient ways, of course, in implementing given binary Boolean function using CNN, the above procedure, however, is simple and works in case of *any local Boolean function*. Hence, the extended CNN cell is universal for implementing any cellular automaton specified by any local Boolean transition rules.

A more complex, more efficient procedure is shown next.

7.6 Minimizing the number of templates when implementing any possible Boolean transition function

The next procedure shows a more efficient, slightly more complex procedure. The number of templates to be used are generally much smaller than the brute force method described in the previous section.

We use again a restricted-weight threshold unit, however, with more possible weight variables[2] in the bias/threshold term z (indeed z=-8, -7, ..., -1, 0, 1, 2, ..., 8) and use any specified two-input logic function Θ, instead of a single one (OR or AND).

Suppose again the state transition rule to be implemented is the neighborhood Boolean function Y.

$$Y = F(u_1, u_1, u_3, ..., u_9)$$

i.e., F is a Boolean function of 9 variables, it can be defined by the 512 bits (as we shown).

We are looking for the solution as a sequence of "ballterms" $b^{(k)}$, k=0, 1, 2, ..., M, implemented with restricted-weight threshold logic (i.e., equivalent templates) and corresponding two-input logic operations Θ^k which will generate F in M steps. All the Boolean functions can be defined by a 2^N-tuple as a response (TRUE or FALSE) to all possible N-tuple input. In our case N = 9. Hence, F is defined as a 512-tuple (there are $2^{512} \approx 10^{154}$ such 512-tuple, hence, different Boolean functions F).

F is generated as follows:

$$f^{(0)} := b^{(0)}$$

$$f^{(1)} := f^{(0)}\, \Theta^{(1)}\, b^{(1)}$$

$$f^{(2)} := f^{(1)}\, \Theta^{(2)}\, b^{(2)}$$

...

$$f^{(M)} := f^{(M-1)}\, \Theta^{(M)} b^{(M)}$$

where $\Theta^{(k)} \in L$ (one of the 16 two-input, one output logic functions).

To calculate the consecutive terms $b^{(k)}\Theta^{(k)}$ (and here $f^{(k)}$) we need *a distance calculation unit* of two N-tuples (u and v) where the distance is calculated as follows

$$\text{dist}(u, v) = \sum_{i=0}^{N-1} u_i \oplus v_i$$

where ⊕ denotes the XOR operation.

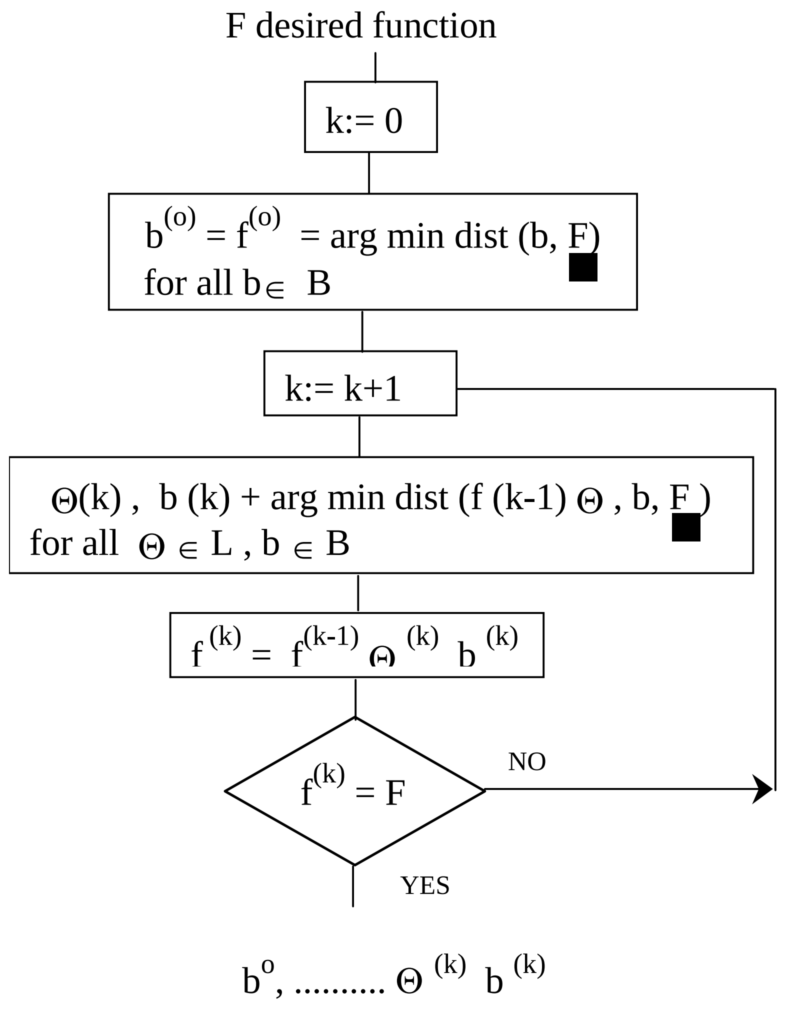
Clearly, using an XOR local logical unit and a few local logic memory units in a cell, this distance (dist (u, v)) calculation can be computed in about N steps.

[2] a function b defined by weights $w_1, w_2, ..., w_9$ and is denoted by $b(w_1, w_1, ..., w_9)$, z.

The distance of two Boolean functions, f and g, of N variables can be calculated similarly

$$\mathrm{dist}_F(f, g) = \sum_{u=0}^{2^N-1} f(u) \oplus g(u); \quad f, g \in F$$

where 2^N XOR operations ($N = 9 \rightarrow 2^N = 512$) are needed.



F desired function
k:= 0
b(o) = f(o) = arg min dist (b, F) for all b ∈ B
k:= k+1
Θ(k) , b (k) + arg min dist (f (k-1) Θ , b, F) for all Θ ∈ L , b ∈ B
f (k) = f(k-1) Θ (k) b (k)
f(k) = F
NO
YES
b°, ... Θ (k) b (k)

Figure 8

The greedy algorithm defined by the flow chart in Fig.8 calculates the consecutive $b^{(k)}$ $\Theta^{(k)}$ functions. $b^{(k)}$ are chosen from the set B. Set B contains all the Boolean functions which can be implemented with the restricted weight values. If N=9, there are $118098=3^9$x6 elements of B (we choose six z values between -8 and +8).

A ballterm b is represented by the 9 feedforward template element values and the z value, denoted by

$b\,(b_1\, b_2\, b_3\, b_4\, b_5\, b_6\, b_7\, b_8\, b_9\,),\ z$

where the last value is the bias value and the order is the same as shown before (the A template has a central nonzero element of 1).

The reasoning behind the algorithm is as follows (search algorithms are denoted by ν in Fig.8.)

- in the first search algorithm, we find $b^{(0)}$ of minimum distance from the prescribed F.
- in the next iterative search algorithm, we test all the possible combinations of the restricted weight functions ($\in$B) and the two-input one-output logic function ($\in$L) to find the best combination, it will modify the previously composed function $f^{(k-1)}$ to $f^{(k)}$ which will be of minimal distance to F.

It is possible to proof that this algorithm converges and, in the worst case, will not result in more terms than the minterm (or maxterm) algorithm shown in the previous section[3].

Example--game of life

This famous problem, with a single Boolean output value, is a linearly non-separable problem. Hence, it cannot be implemented by a single template. The algorithm in Fig.8 results in just 2 terms:

$$b^{(0)} = b(-1, -1, -1, -1, 0, -1, -1, -1, -1), +1$$

$$b^{(1)} = b(+1, +1, +1, +1, +1, +1, +1, +1, +1), -4$$

$$\Theta^{(1)} = \text{AND}$$

Hence, we can implement the game of life with a cell of Fig.7 containing an AND local logical unit, and the two templates which implement the “ballterms” $b^{(0)}$ and $b^{(1)}$ are

$$A = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix};\ B^{(0)} = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 0 & -1 \\ -1 & -1 & -1 \end{bmatrix};\ z = +1$$

and

$$A = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix};\ B^{(1)} = \begin{bmatrix} +1 & +1 & +1 \\ +1 & +1 & +1 \\ +1 & +1 & +1 \end{bmatrix};\ z = -4$$

respectively.

[3] K. R. Crounse, E. L. Fung and L. O. Chua, , Efficient implementation of neighborhood logic for cellular automata via the cellular neural network universal machine, IEEE Trans. CAS-I, Vol. 44, pp. 355-361, 1997

7.7 Analog-to-digital array converter

SUBROUTINE ADARRAY(., ., .)

ADARRAY(P, n, B[0, n-1])

P:	positive image $0 \le p_{ij} \le 1$
n:	integer, number of bits
B[0, n-1]:	$B_{ij}[k]$: The value of the k-th bit $\in \{0, 1\}$,

k=0,1,2, ..., n-1.

Global task Array-type analog to digital converter.

Given a signal array P at a given time instant t_0, i.e., $P = P(t)|_{t=t0}$.

$P = [p_{ij}]$, i=1,2,..., M; j=1,2, ..., N. $0 \le p_{ij} \le 1$.

Compute the representation of the real (analog) values

p_{ij}: $B_{ij}[k]$, k=0, 1, 2, ..., n-1.

The algorithm

The algorithm (a well known method) is given for a single cell, all cells are computing fully parallel, without interaction.

Given: p, $0 \le p_{ij} \le 1$ real and n, integer, r: real, b: binary,

```
 let : r(-1):=p      and      b(-1):=1
FOR i:=0 step 1 until i<n      DO

        begin
                r(i):=2r(i-1) - b(i-1)
                b(i):=sgn(r(i))
                B(i):=bconvert (b(i))
        end
```

where “bconvert” (binary converter) is a function with input {-1, +1} and output {0,1} which represent logic LOW and HIGH. B(i) are the sequence of the output bits.

Example

Convert the value p=0.6875=$\frac{1}{2}+0.\frac{1}{4}+\frac{1}{8}+\frac{1}{16}$ (i.e., the code B() is: 1011). The consecutive steps of the algorithm are as follows

```
r(-1)=0.6875;    b(-1) =1

i=0
begin   r(0) = 2x0.6875-1= 0.375
            b(0) = sgn(0.375) = 1
            B(0) = bconvert(1)=          [1]
end

i=1
begin   r(1) = 2x0.375-1= -0.25
            b(1) = sgn(-0.25) = -1
            B(1) = bconvert(-1)=         [0]
end

i=2
begin   r(2) = 2x(-0.25) + 1= 0.5
            b(2) = sgn(0.5) = 1
            B(2) = bconvert(1)=          [1]
end

i=3
begin   r(3) = 2x0.5-1= 0
            b(3) = sgn(0) = 1
            B(3) = bconvert(1)=          [1]
end
```

CNN implementation

A: hardwired components

To implement this algorithm via CNN we need some additional components in each cell. For the time being, we suppose that each template is implemented by a single CNN standard cell, as a component, and the whole array is hardwired form the components. An extended cell (type 2) is shown below.

Extended cell 2

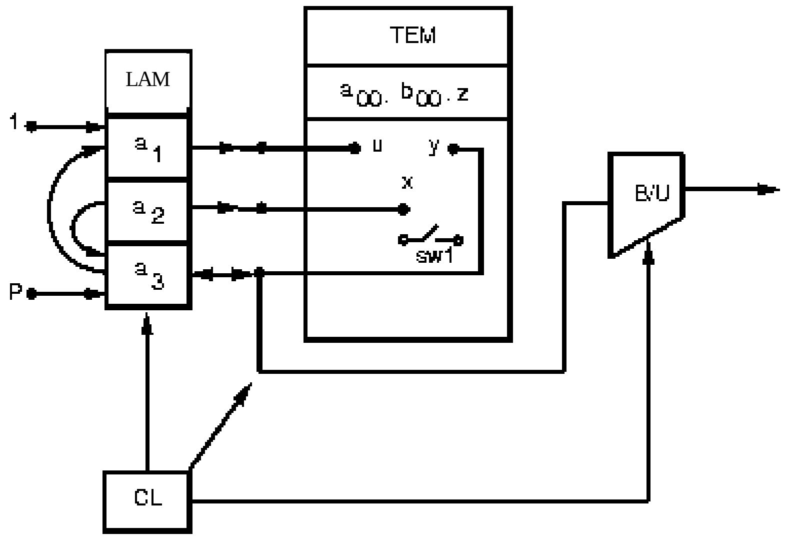


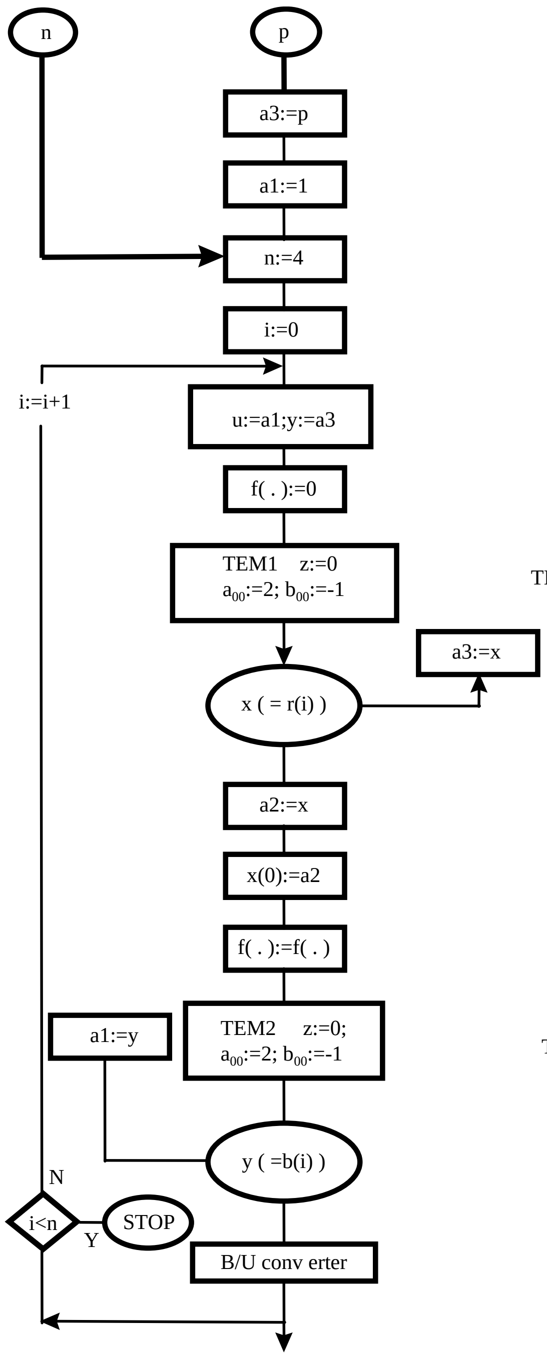
Fig.8 The extended cell 2. In addition to the CNN cell which may have a switch, we have 2 new components: an analog storage device, a local analog memory (LAM) and a bipolar to unipolar converter (B/U).

The new component in the Extended Cell 2 are:

- in some CNN standard cells (type 2), for the time being we consider them as separate components, there is a *switch* SW1 which, if it is OFF, set the value of the standard nonlinearity of the cell to zero, i.e., if SWI: OFF then f(.)=0; we suppose that if SW1=OFF then the input and output of the cell can be specified and the value at the state will be the outcome,
- an analog memory unit LAM (*local analog memory*), in this case with 3 storage places,
- a binary converter B/U (denoted by bconvert(.)), converting a bipolar {-1,1} analog signal into a unipolar {0, 1}-{LOW, HIGH} logic bit.

Suppose we place the Extended Cell 2 in a CNN array, then we can design the flow diagram of the A/D algorithm. This flow diagram is shown in Figure 9. On the same figure we show parallel, the program of the algorithm implementing our A/DARRAY(.) subroutine, for a single cell.

Here we suppose that this program is hardwired, i.e., the clock signals activate the subsequent units according to a predefined sequence.



```
FUNCTION ADARRAY;
USE(TEM0);
  N=4;
LAM3:=  p ;
LAM1:= 1 ;
REPEAT i:= 0 TO n BY 1
SW1:=OFF;
TEM0(LAM1,LAM2,LAM3, 5, -1);
LAM3:=LAM2;
SW1 := ON;
TEM0(LAM1,LAM2,LAM1, 5, -1);
B[ i ] := bconvert (LAM1);
END REPEAT;
END
```

Figure 9