



# Fondamenti di Elaborazione di Immagini

## Topologia Digitale

Raffaele Cappelli  
[raffaele.cappelli@unibo.it](mailto:raffaele.cappelli@unibo.it)

# Contenuti

- Metriche e distanze
- Trasformata distanza
- Contorno di un oggetto
- Etichettatura delle componenti connesse
- Thinning

# Topologia digitale

- Disciplina che studia **proprietà e caratteristiche topologiche** delle immagini (es. componenti connesse, bordi di un oggetto, ...)
- Considera **immagini digitali binarie** (2 soli livelli di grigio)
  - Foreground (in genere 255 o 0, nel seguito indicato con *foreground*)
  - Background (in genere 0 o 255, nel seguito indicato come *≠foreground*)
- Sia  $F$  l'insieme di tutti i pixel di foreground e  $F^*$  l'insieme di quelli di background di un'immagine  $Img$ :
  - $F = \{\mathbf{p} \mid \mathbf{p} = [x, y]^T, Img[y, x] = foreground\}$
  - $F^* = \{\mathbf{p} \mid \mathbf{p} = [x, y]^T, Img[y, x] \neq foreground\}$

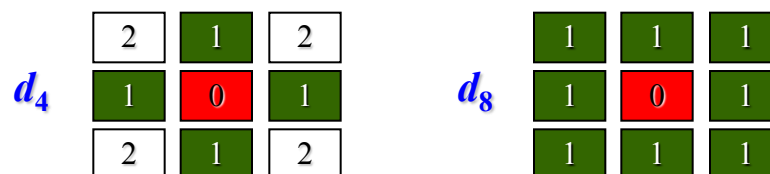
# Topologia digitale – Metriche e distanze

- Le **metriche** discrete più comuni sono basate su distanza  $d_4$  (city-block) e  $d_8$  (chessboard).
- I **vicini** di un pixel  $\mathbf{p}$  sono quelli aventi distanza unitaria da  $\mathbf{p}$ .
- Un **percorso** di lunghezza  $n$  da  $\mathbf{p}$  a  $\mathbf{q}$  è una sequenza di pixel  $\{\mathbf{p}_0=\mathbf{p}, \mathbf{p}_1, \dots, \mathbf{p}_n=\mathbf{q}\}$  tale che, in accordo con la metrica scelta,  $\mathbf{p}_i$  è un vicino di  $\mathbf{p}_{i+1}$ ,  $0 \leq i < n$ .
- Una **componente connessa** è un sottoinsieme di  $F$  (o di  $F^*$ ) tale che, presi due qualsiasi dei suoi pixel, esiste tra questi un percorso appartenente a  $F$  (o  $F^*$ ). A seconda della metrica adottata si parla di 4-connessione o di 8-connessione.

$$\mathbf{p} = [x_p, y_p]^T, \mathbf{q} = [x_q, y_q]^T$$

$$d_4(\mathbf{p}, \mathbf{q}) = |x_q - x_p| + |y_q - y_p|$$

$$d_8(\mathbf{p}, \mathbf{q}) = \max\{|x_q - x_p|, |y_q - y_p|\}$$



Vicini di un pixel  $\mathbf{p}$  secondo le due metriche



Percorso ( $d_4$ )



Percorso ( $d_8$ )



Due componenti connesse secondo  $d_4$

Un'unica componente connessa secondo  $d_8$

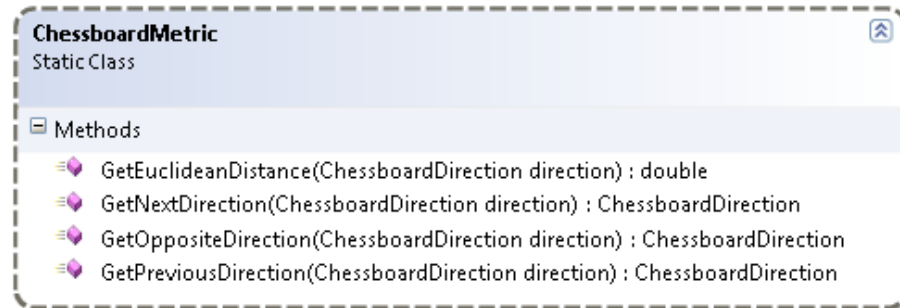
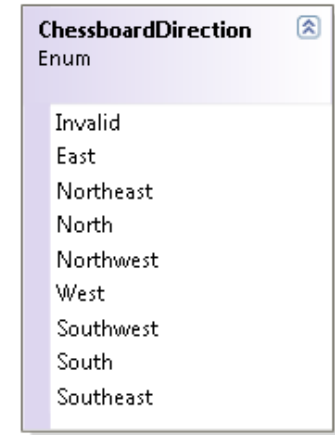
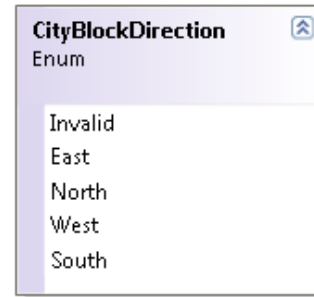
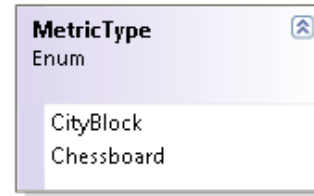
# Metriche nella libreria

## ■ Enum

- Tipi di metriche
- Direzioni per ciascuna delle due metriche considerate

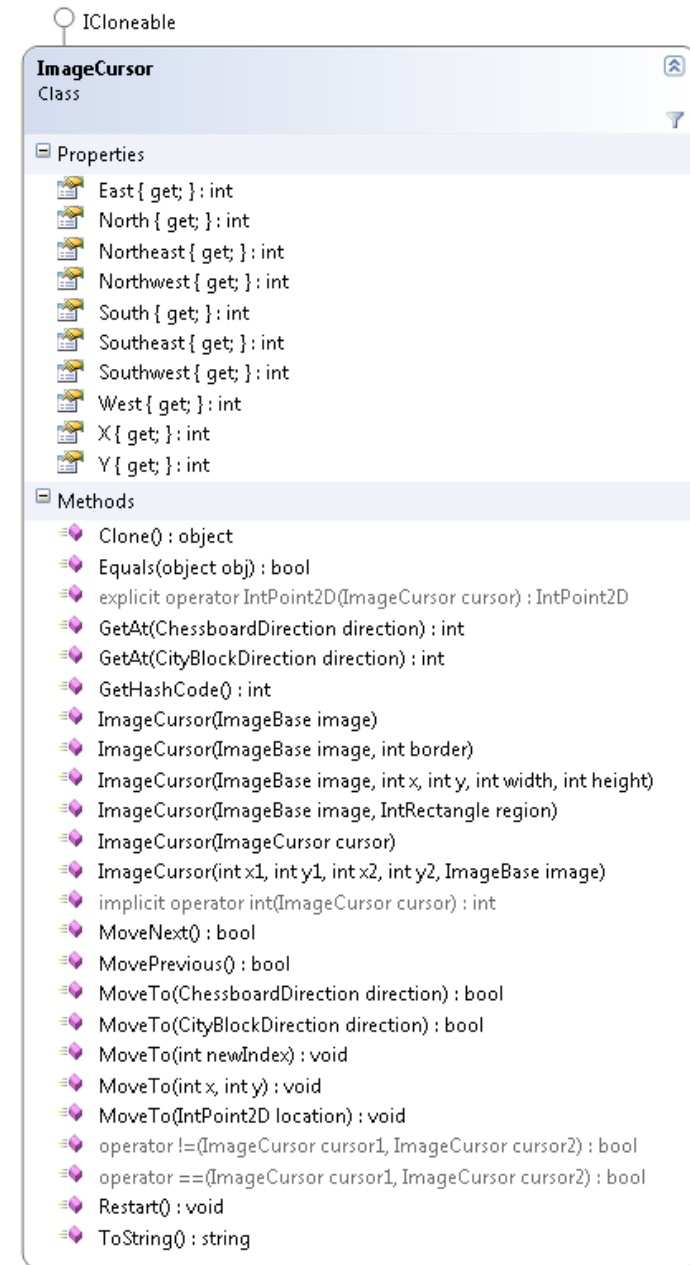
## ■ Classi statiche di utilità

- Per ciascuna delle metriche considerate
- Contengono metodi statici per ottenere la direzione opposta, seguente (o precedente), in senso antiorario, rispetto a una direzione data



# Una classe “cursore”

- L'immagine è memorizzata come un array monodimensionale
  - L'implementazione di algoritmi di topologia digitale risulterebbe semplificata **accedendo all'immagine come se fosse una matrice**, ma sarebbe piuttosto **inefficiente** (una moltiplicazione e una somma per ricavare l'offset ogni volta che si accede a un pixel)
  - Un buon compromesso fra semplicità nella stesura degli algoritmi ed efficienza può essere ottenuto implementando una classe “cursore”, che consenta di **scorrere i pixel dell'immagine**, eventualmente escludendo i bordi, e fornisca **proprietà e metodi per accedere ai pixel vicini** secondo una delle metriche  $d_4$  o  $d_8$



# ImageCursor - Esempi

```
// Esempio: conteggio dei pixel di foreground
var p = new ImageCursor(img); // Predispone il cursore per
                                // scandire tutti i pixel di img

int numFGPixels = 0;
do
{
    if (img[p] == foreground) // Il cursore può direttamente essere usato
        numFGPixels++;        // come indice di un pixel dell'immagine
} while (p.MoveNext());
MessageBox.Show(String.Format("Pixel di foreground = {0}", numFGPixels));

// Esempio: conteggio dei pixel isolati (metrica d4)
var p = new ImageCursor(img, 1); // Esclude un pixel di bordo
int numIsolatedPixels = 0;
do
{
    if (img[p]==foreground && img[p.North]!=foreground &&
        img[p.East]!=foreground && img[p.South] != foreground &&
        img[p.West] != foreground)
        numIsolatedPixels++;
} while (p.MoveNext());
MessageBox.Show(String.Format("Pixel isolati = {0}", numIsolatedPixels));
```

# Trasformata distanza

## ■ Definizione

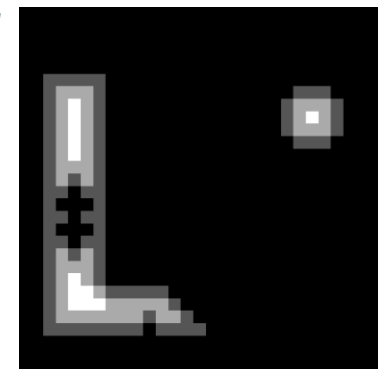
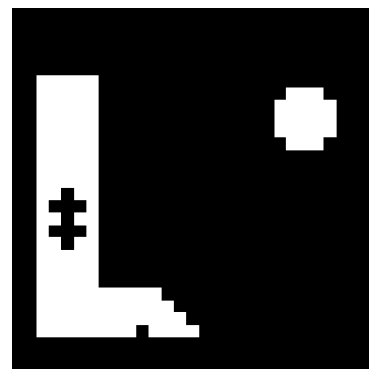
- La trasformata distanza di  $F$  rispetto a  $F^*$  è una replica di  $F$  in cui i pixel sono etichettati con il valore della loro distanza da  $F^*$ , calcolata secondo una data metrica.

## ■ Esempi (metrica $d_4$ )

0	0	0	0	0	0	0	0
0	255	255	255	255	255	255	0
0	255	255	255	255	255	255	0
0	255	255	255	255	255	255	0
0	255	255	255	255	255	255	0
0	255	255	255	255	255	255	0
0	255	255	255	255	255	255	0
0	0	0	0	0	0	0	0



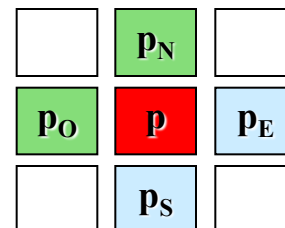
0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	0
0	1	2	2	2	2	1	0
0	1	2	3	3	2	1	0
0	1	2	2	2	2	1	0
0	1	1	1	1	1	1	0
0	0	0	0	0	0	0	0





# Calcolo della trasformata distanza

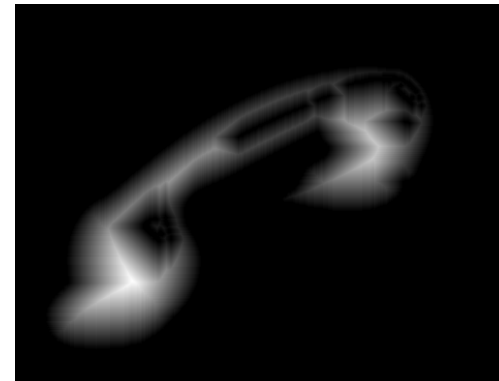
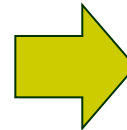
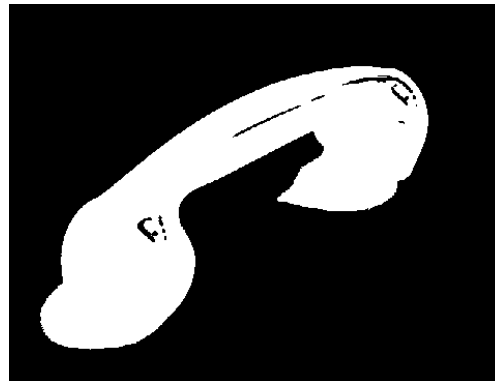
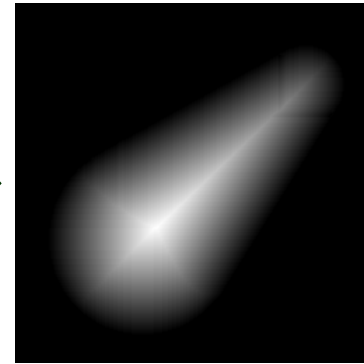
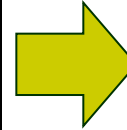
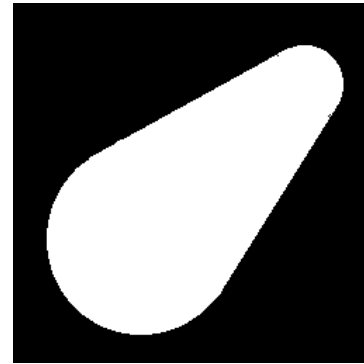
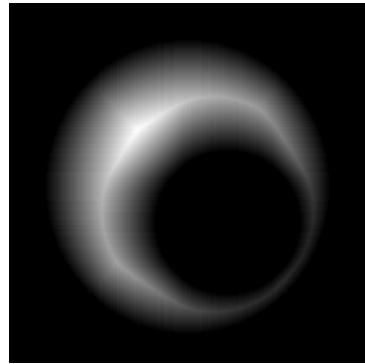
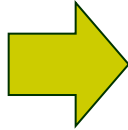
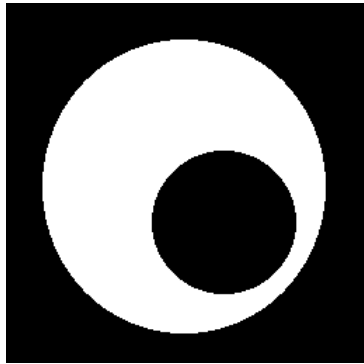
- Si considera il caso della metrica  $d_4$ .
  - La trasformata può essere calcolata con due semplici scansioni dell'immagine:
    - una diretta (dall'alto verso il basso da sinistra verso destra),
    - una inversa (dal basso verso l'alto da destra verso sinistra).
    - Durante le scansioni i pixel di  $F$  vengono trasformati; i pixel di  $F^*$  sono posti a zero (distanza nulla).
  - $\forall \mathbf{p} \in F$ , il valore trasformato  $\text{Img}'[\mathbf{p}]$  è calcolato come segue:
    - Scansione diretta: la distanza viene propagata dai vicini in alto a sinistra (che sono già stati considerati).
      - $\text{Img}'[\mathbf{p}] = \min\{\text{Img}'[\mathbf{p}_O], \text{Img}'[\mathbf{p}_N]\} + 1$
    - Scansione inversa: la distanza viene aggiornata tenendo conto anche dei percorsi verso il basso e a destra.
      - $\text{Img}'[\mathbf{p}] = \min\{\text{Img}'[\mathbf{p}_E]+1, \text{Img}'[\mathbf{p}_S]+1, \text{Img}'[\mathbf{p}]\}$



# Trasformata distanza – Implementazione

```
Result = new Image<int>(InputImage.Width, InputImage.Height);  
var r = Result;  
var cursor = new ImageCursor(r, 1);  
  
if (Metric == MetricType.CityBlock)  
{  
    do // Scansione diretta  
    {  
        if (InputImage[cursor] == Foreground)  
        {  
            r[cursor] = Min(r[cursor.West], r[cursor.North]) + 1;  
        }  
    } while (cursor.MoveNext());  
  
    do // Scansione inversa  
    {  
        if (InputImage[cursor] == Foreground)  
        {  
            r[cursor] = Min(r[cursor.East] + 1, r[cursor.South] + 1, r[cursor]);  
        }  
    } while (cursor.MovePrevious());  
}  
else  
{ ... }
```

# Trasformata distanza – Esempi



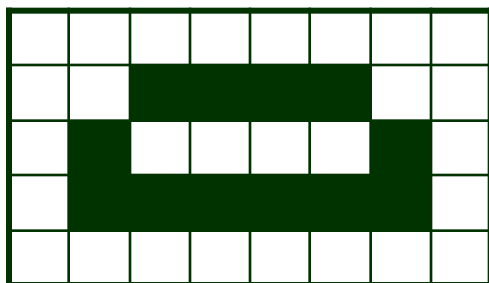
# Trasformata distanza – Applicazioni

- Può essere utilizzata per la misurazione geometrica di: lunghezze, spessori, ...
- La trasformata distanza produce massimi locali di intensità in corrispondenza degli assi mediani dell'oggetto (scheletro).
  - Nel caso di oggetti allungati la trasformata può essere utilizzata per algoritmi di assottigliamento o thinning.
- Applicazioni più avanzate della trasformata (o di sue varianti) includono:
  - Template matching
  - Robotica (ricerca direzione di movimento ottimale, aggiramento ostacoli, ...)

# Estrazione del contorno

## ■ Definizioni

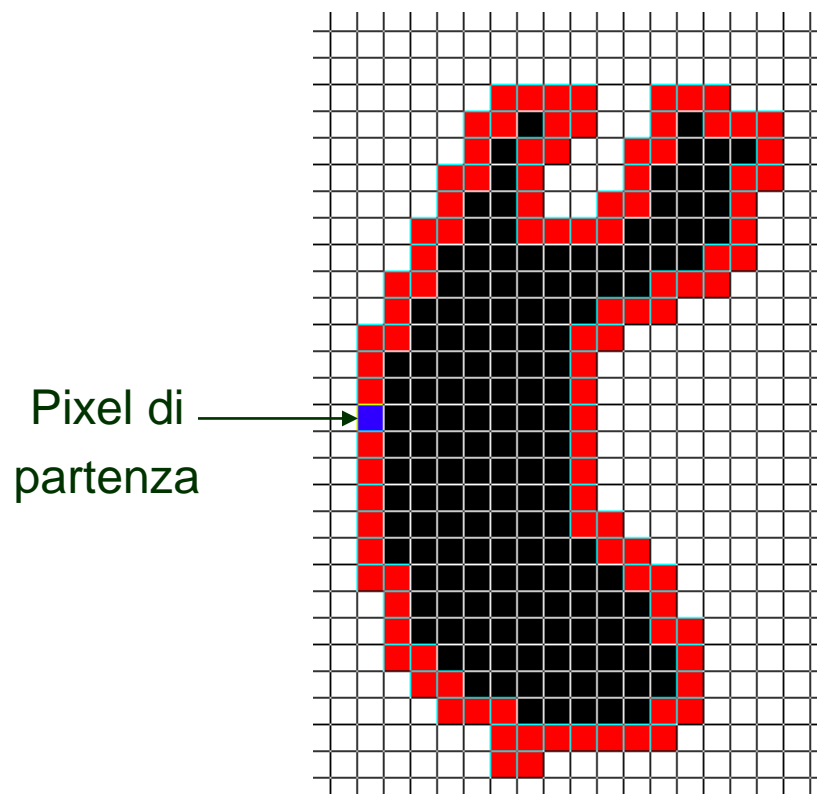
- Il contorno di  $F$  è costituito dall'insieme di pixel che hanno distanza unitaria da  $F^*$  secondo la metrica adottata per  $F^*$ .
- Scegliere la metrica  $d_4$  per  $F$  implica necessariamente utilizzare  $d_8$  per  $F^*$  e viceversa. In caso contrario si avrebbero “paradossi” topologici.



Paradosso: scegliendo  $d_8$  sia per  $F$  che per  $F^*$ , la curva chiusa in figura non separerebbe in due regioni lo sfondo.

## Estrazione del contorno (2)

- Per l'estrazione del contorno di un oggetto (ovvero della sequenza ordinata di pixel che lo compongono) viene normalmente impiegata una semplice tecnica di inseguimento che percorre il bordo sempre nella stessa direzione fino a incontrare il pixel di partenza.

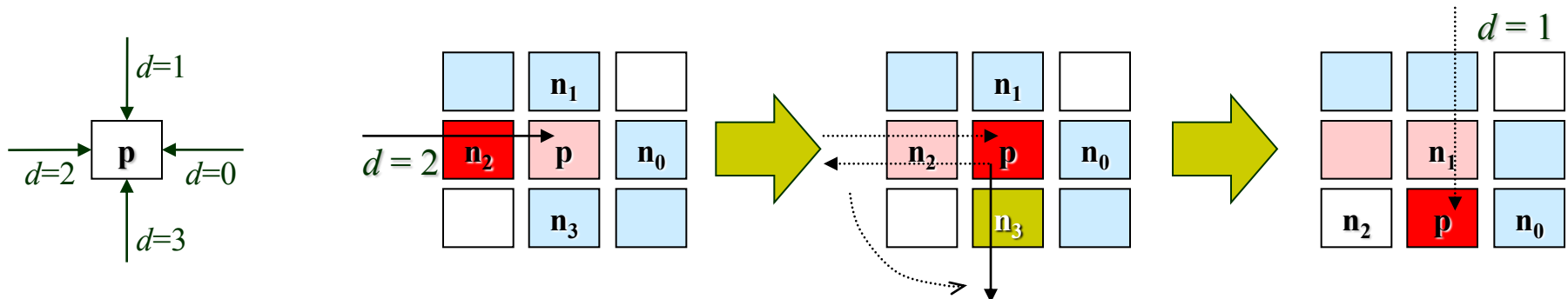


Contorno dell'oggetto se si adotta  $d_4$  per  $F$  (e quindi  $d_8$  per  $F^*$ ):  
i pixel evidenziati sono i soli pixel di  $F$  per cui la distanza  $d_8$  da  $F^*$  vale 1.

# Estrazione del contorno (3)

## ■ Algoritmo

- Si considera il caso con metrica  $d_4$  per  $F$  (e quindi  $d_8$  per  $F^*$ ) e verso di percorrenza antioraria del bordo.
- Sia  $p$  il pixel corrente (il pixel iniziale può essere uno qualunque dei pixel di bordo): il prossimo pixel da visitare è uno dei 4 vicini a  $p$ .
- Sia  $d$  la direzione corrente di inseguimento: rispetto alla figura la direzione è indicata dall'indice del pixel da cui si è giunti a  $p$ :
  - allora il prossimo pixel da visitare è il primo tra i seguenti che appartiene a  $F$ :  
 $\{ n_{(d+1) \bmod 4}, n_{(d+2) \bmod 4}, n_{(d+3) \bmod 4}, n_{(d+4) \bmod 4} \}$ .
  - L'idea è consiste infatti nel prendere il primo pixel di bordo nella direzione il più possibile "chiusa" in senso antiorario rispetto a  $d+180^\circ$ .



# Codifica del contorno

- Una volta estratto, il bordo può essere codificato in vari modi:
  - Semplice lista delle coordinate  $x,y$ 
    - Non è certo la soluzione più vantaggiosa
  - Lista degli indici dei pixel (intesi come offset nell'array monodimensionale)
    - Scelta poco conveniente: è dipendente dalla larghezza dell'immagine
  - Chain code
    - Invece di memorizzare la lista di coordinate dei punti di bordo, si memorizzano **le direzioni** (o meglio gli indici delle direzioni) che permettono di generare il contorno a partire da un pixel prefissato.
  - Approssimazione poligonale/spline
    - Si sotto-campionano i pixel lungo il bordo (prendendo un punto ogni  $n$ ) e si costruisce, a partire da tali pixel, una poligonale o una curva spline.



# Estrazione del contorno – Implementazione

```
// Parametri:  
// pixelStart : punto iniziale (ImageCursor)  
// direction : direzione iniziale [0..3] (CityBlockDirection)  
// contour : contorno, memorizzato come Chain Code, con un metodo Add()  
// image : l'immagine (Image<byte>)  
  
var cursor = new ImageCursor(pixelStart);  
do  
{  
    for (int j = 1; j <= 4; j++)  
    {  
        direction = CityBlockMetric.GetNextDirection(direction);  
        if (image[cursor.GetAt(direction)] == Foreground)  
        {  
            break;  
        }  
    }  
    contour.Add(direction); // nuova direzione aggiunta al contorno  
    cursor.MoveTo(direction); // si sposta seguendo la direzione  
    direction = CityBlockMetric.GetOppositeDirection(direction); // direzione  
                                                                    // rispetto al pixel precedente  
} while (cursor != pixelStart); // si ferma quando incontra il pixel iniziale
```

# Etichettatura delle componenti connesse

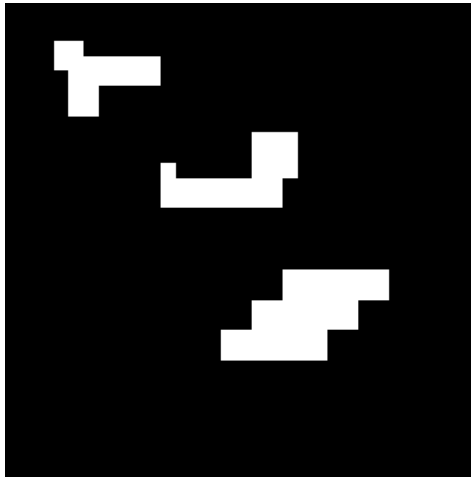
## ■ In cosa consiste

- Individuare automaticamente le diverse componenti connesse in un'immagine, assegnando loro etichette (tipicamente numeriche)
- Procedura molto importante nell'analisi automatica di immagini

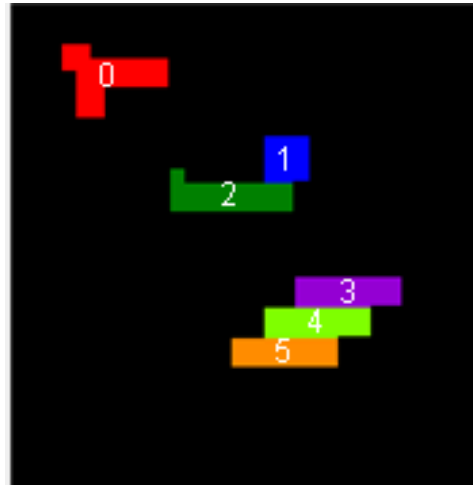
## ■ Algoritmo comunemente utilizzato:

- 1) Si scandisce l'immagine e, per ogni pixel di foreground  $p$ , si considerano i pixel di foreground vicini già visitati:
  - se nessuno è etichettato, si assegna una nuova etichetta a  $p$
  - se uno è etichettato, si assegna a  $p$  la stessa etichetta
  - se più di uno è etichettato, si assegna a  $p$  una delle etichette e si annotano le equivalenze
- 2) Si definisce un'unica etichetta per ogni insieme di etichette marcate come equivalenti e si effettua una seconda scansione assegnando le etichette finali

# Etichettatura – Esempio



Immagine



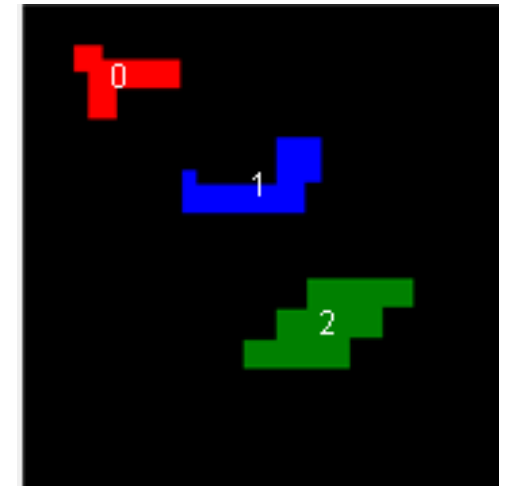
Prima scansione

Equivalenze

$2=1 \rightarrow 1$

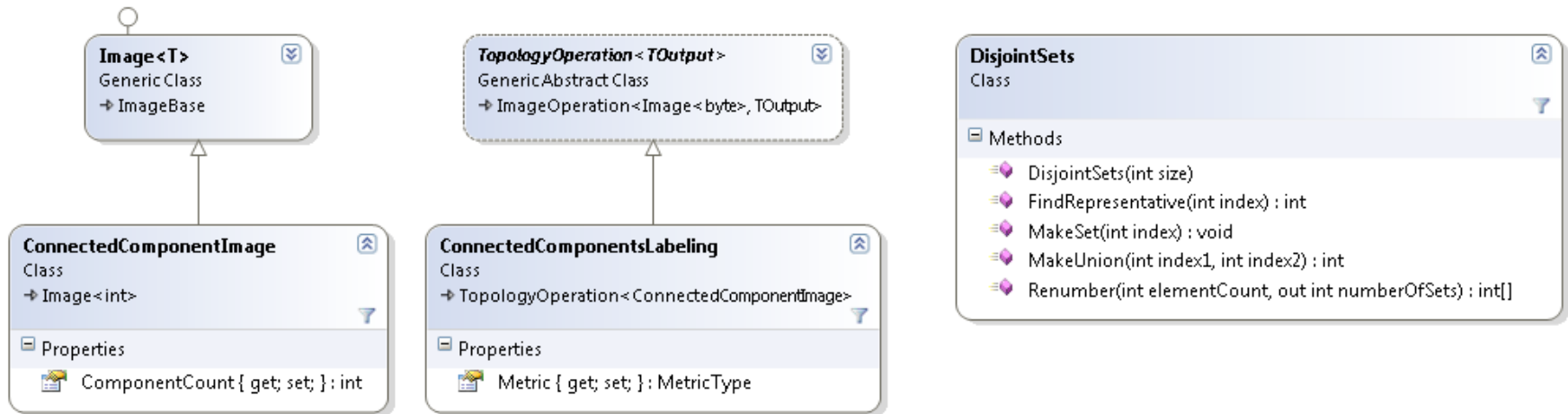
$4=3 \rightarrow 2$

$5=3 \rightarrow 2$



Seconda scansione

# Etichettatura – Classi della libreria



## ■ ConnectedComponentImage

- Derivata da **Image<int>**: per ogni pixel memorizza l'indice della componente connessa a cui appartiene; inoltre proprietà **ComponentCount**: numero totale componenti connesse.

## ■ DisjointSets

- Struttura dati utile per gestire insiemi disgiunti di interi. Rappresenta i valori come una foresta di alberi (ogni albero è un insieme). La radice di ciascun albero è il rappresentante dell'insieme.
  - **MakeSet(x)**: crea un nuovo insieme contenente il solo elemento x.
  - **MakeUnion(x,y)**: fonde gli insiemi contenenti i due elementi x e y.
  - **FindRepresentative(x)**: restituisce il rappresentante dell'insieme a cui appartiene x.
  - **ReNUMBER**: rende i rappresentanti degli insiemi numeri consecutivi.

# Etichettatura – Implementazione di base

```

Result = new ConnectedComponentImage(InputImage.Width, InputImage.Height, -1);
var cursor = new ImageCursor(InputImage, 1); // per semplicità ignora i bordi (1 pixel)
int[] neighborLabels = new int[Metric == MetricType.CityBlock ? 2 : 4];
int nextLabel = 0;
var equivalences = new DisjointSets(InputImage.PixelCount);
do { // prima scansione
    if (InputImage[cursor] == Foreground)
    {
        int labelCount = 0;
        if (Result[cursor.West] >= 0) neighborLabels[labelCount++] = Result[cursor.West];
        if (Result[cursor.North] >= 0) neighborLabels[labelCount++] = Result[cursor.North];
        if (Metric == MetricType.Chessboard)
        {
            // anche le diagonali
            if (Result[cursor.Northwest] >= 0) neighborLabels[labelCount++] = Result[cursor.Northwest];
            if (Result[cursor.Northeast] >= 0) neighborLabels[labelCount++] = Result[cursor.Northeast];
        }
        if (labelCount == 0)
        {
            equivalences.MakeSet(nextLabel); // crea un nuovo set
            Result[cursor] = nextLabel++; // le etichette iniziano da 0
        }
        else
        {
            int l = Result[cursor] = neighborLabels[0]; // seleziona la prima
            for (int i = 1; i < labelCount; i++) // equivalenze
                if (neighborLabels[i] != l)
                    equivalences.MakeUnion(neighborLabels[i], l); // le rende equivalenti
        }
    }
} while (cursor.MoveNext());

```

# Etichettatura – Implementazione di base (2)

```
// ...continua dal lucido precedente

// rende le etichette numeri consecutivi
int totalLabels;
int[] corresp = equivalences.Renumber(nextLabel, out totalLabels);

// seconda e ultima scansione
cursor.Restart();
do
{
    int l = Result[cursor];
    if (l >= 0)
    {
        Result[cursor] = corresp[l];
    }
} while (cursor.MoveNext());
Result.ComponentCount = totalLabels;
```

# Etichettatura componenti connesse – Esempi

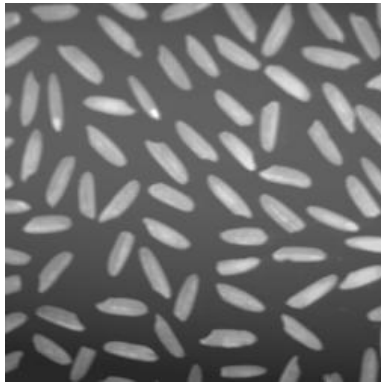
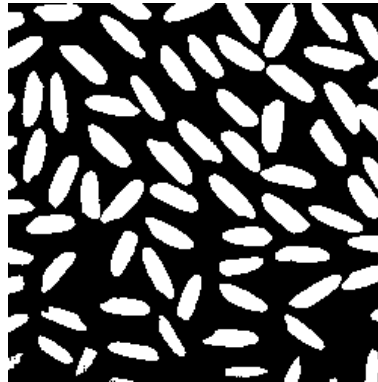
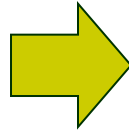
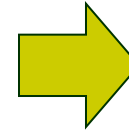


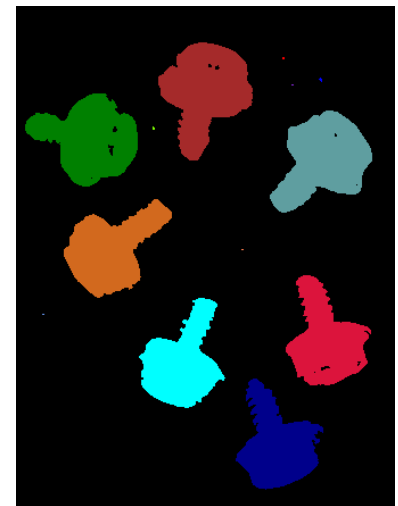
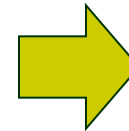
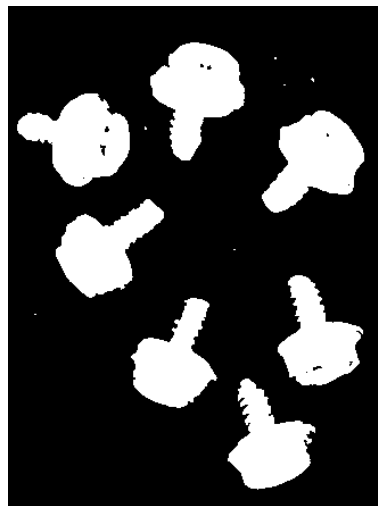
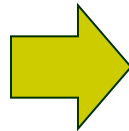
Immagine originale



Binarizzazione



Etichettatura



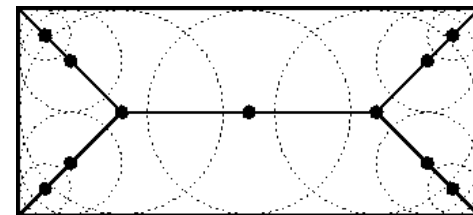
# Scheletro e Thinning

## ■ Definizioni



### □ Scheletro di un'immagine binaria (skeleton)

- Preserva le **caratteristiche topologiche** dell'oggetto
- Può essere definito come il **luogo dei centri dei cerchi, tangenti in almeno due punti**, che sono completamente contenuti nel foreground
- Gli algoritmi che cercano di implementare direttamente tale definizione sono computazionalmente assai inefficienti
- In genere si cerca di ottenere lo scheletro mediante algoritmi di thinning



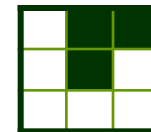
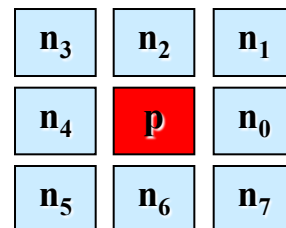
### □ Thinning (assottigliamento)

- Gli algoritmi di thinning procedono tipicamente in modo iterativo
- Ad ogni iterazione eliminano da  $F$  (attribuendoli a  $F^*$ ) pixel di bordo che la cui cancellazione non altera la topologia locale, ossia:
  - Non rimuovono pixel terminali
  - Non eliminano pixel di connessione
  - Non causano erosione eccessiva
- Le iterazioni proseguono fino a quando non vi sono più pixel da eliminare

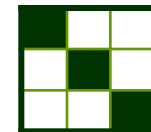


# Thinning – Algoritmo di Hilditch

- Uno dei più noti metodi di thinning:
  - Sia  $A(p)$  il numero di transizioni *Background* → *Foreground* nella sequenza ordinata:  
 $n_0, n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_0$
  - Sia  $B(p)$  il numero di pixel appartenenti a  $F$  nell'8-intorno di  $p$ .
- Ad ogni passo:
  - l'algoritmo considera tutti i pixel in modo parallelo (l'operazione su un pixel non dipende da quelli già esaminati)
  - i pixel  $p$  che rispettano le 4 condizioni a,b,c,d sono cancellati
- L'algoritmo termina quando nessun nuovo pixel può essere cancellato



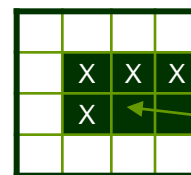
$A(p)=1, B(p)=2$



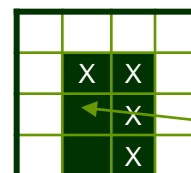
$A(p)=2, B(p)=2$

## Condizioni

- a)  $2 \leq B(p) \leq 6$
- b)  $A(p) = 1$
- c)  $n_2=background$  or  $n_0=background$  or  $n_4=background$  or  $A(n_2) < 1$
- d)  $n_2=background$  or  $n_0=background$  or  $n_6=background$  or  $A(n_0) < 1$



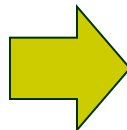
Pixel non eliminato per la condizione c)



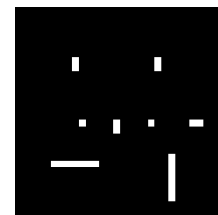
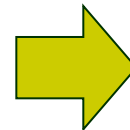
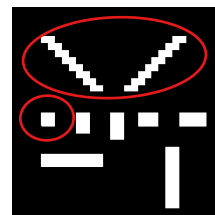
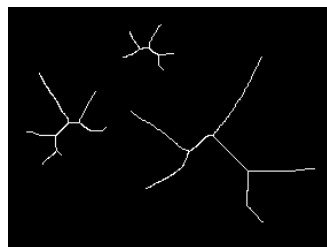
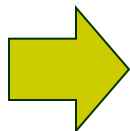
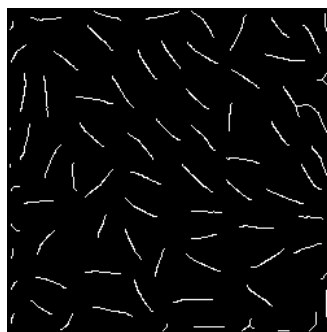
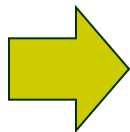
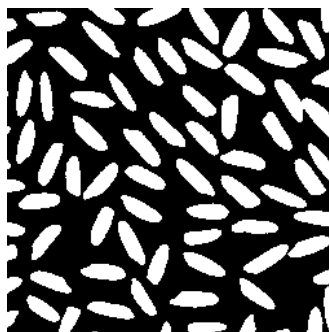
Pixel non eliminato per la condizione d)

# Algoritmo di Hilditch – Esempi

*Cantami, o Diva, del Pelide Achille l'ira  
funesta che infiniti addusse lutti agli  
Achei, molte anzi tempo all'Orco generose  
travolse alme d'eroi,*



*Cantami, o Diva, del Pelide Achille l'ira  
funesta che infiniti addusse lutti agli  
Achei, molte anzi tempo all'Orco generose  
travolse alme d'eroi,*



L'algoritmo non è perfetto: alcuni pattern  
sono erosi completamente (o quasi)

# Algoritmo di Hilditch – Implementazione di base

```

Image<byte> img,img1; // img: immagine di input; img1: immagine di appoggio
ImageCursor p1,p2; // Cursori - p1: 1 pixel di bordo, p2: 2 pixel di bordo
Image<byte> imgA,imgB; // Ad ogni iterazione A[p],B[p] calcolati per ogni pixel
int removedPixelCount;
do { // Iterazione: input immagine img output immagine img1
    removedPixelCount = 0;
    p1.Restart(); // calcola imgA[p1] e imgB[p1] per ogni p1
    do {
        imgA[p1] = ...; imgB[p1] = ...;
    } while (p1.MoveNext());
    p2.Restart(); // p2 scandisce l'immagine lasciando 2 pixel di bordo
    do {
        byte pixelValue = img[p2];
        if (pixelValue == Foreground)
        { // se soddisfa tutte le 4 condizioni, va cancellato
            if (...) {
                pixelValue = background;
                removedPixelCount++;
            }
        }
        img1[p2] = pixelValue;
    } while (p2.MoveNext());
    // scambia i riferimenti alle due immagini di lavoro
    var tmp = img1; img1 = img; img = tmp;
} while (removedPixelCount > 0);

```