第七單元 細線化與骨架抽取

Thinning and Skeletonization

陳慶瀚

2004-11-09

定義：*4-connectivity* and *8-connectivity*

─Two pixels are said to be 4-connected to each other if they are neighbors via any of the four cardinal directions (N, E, S, W).

─Two pixels are said to be 8-connected if they are neighbors via any of the eight directions (N, NE, E, SE, S, SW, W, NW).

─An object is said to be 4-connected if any of its pixels can be reached from any other pixel of the same object by traversing via 4-connected pixel pairs.

1. Distance Transform(DT)

DT 是一種應用在二值影像的運算子，其運算結果則為一灰階影像。與一般灰階影像不同，其強度並非表示亮度值，而是表示物件內部每一點與物件邊緣的距離。

設有物件內部的兩點 $p_1=(x_1, y_1)$, 和 $p_2=(x_2, y_2)$， 其距離可以用以下三種距離測度*(distance metrics)*表示：

Euclidean distance:

$$D_e(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

City block distance:

$$D_4(p_1, p_2) = |x_1 - x_2| + |y_1 - y_2|$$

Chessboard distance:

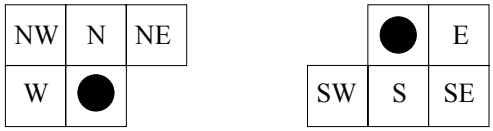
$$D_8(p_1, p_2) = \max(|x_1 - x_2|, |y_1 - y_2|)$$

如果以 1 表示物件像素，0 是背景像素，則 *distance transform* 定義為

對於每一個物件區域的像素，計算其與最近的背景像素的距離，並以此距離值取代原像素值。

計算 ***8-distance transform***

使用兩個 local window 進行 two-pass 的計算：



左邊的 window 用於前向的計算(forward pass), 右邊的 window 則用於後向的計算(backward pass)。

前向計算的順序是由左而右、由上而下，後向計算的順序則正好相反。每一次計算都檢查 4 個方向的最短距離(前向時是 W, NW, N, NE, 後向時則為 E, SE, S, SW)，詳細演算法如下：

1. Initialize the image pixels:

   Label the pixels $d_{x,y} = \begin{cases} 1, & \text{if pixel } x,y \text{ is } \textit{black} \\ 0, & \text{if pixel } x,y \text{ is } \textit{white} \end{cases}$

2. Forward pass:

   For each object pixel: $d_{x,y} = \min\left(d_W, d_{NW}, d_N, d_{NE}\right) + 1$

3. Backward pass:

   For each object pixel: $d_{x,y} = \min\left(d_{x,y}, d_E + 1, d_{SE} + 1, d_S + 1, d_{SW} + 1\right)$

一個 ***8-distance transform*** 執行範例如下：

Original image:

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

前向計算後:

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

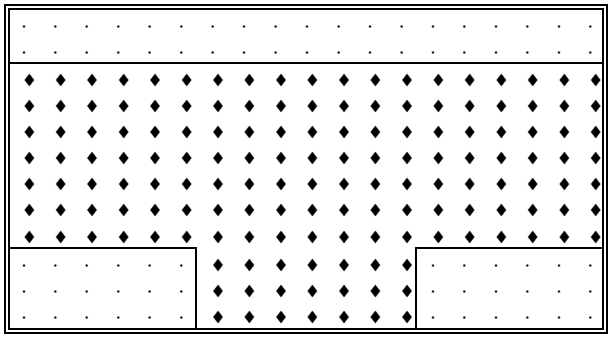
後向計算後:

0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	0
0	1	2	2	1	1	1	0
0	1	2	2	1	0	0	0
0	1	1	1	1	0	0	0
0	0	0	0	0	0	0	0

計算 *4-distance transform*

4-distance transform 也可以使用類似的演算法，只是改成使用兩個方向(前向時是 W 和 N, 後向時則為 E 和 S)即可。.

下圖展示一張二值影像經的 ***distance transform*** 的計算：(a)是原始影像，(b)和(c)分別是 ***4-distance*** 和 ***8-distance*** 轉換的結果。



(a)**Original image**

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
4	**4**	**4**	**4**	**4**	**4**	4	4	4	4	4	4	4	**4**	**4**	**4**	**4**	**4**	**4**
3	3	3	3	3	3	4	**5**	5	5	5	**5**	4	3	3	3	3	3	3
2	2	2	2	2	2	3	4	5	**6**	5	4	3	2	2	2	2	2	2
1	1	1	1	1	1	2	3	4	5	4	3	2	1	1	1	1	1	1
0	0	0	0	0	0	1	2	3	4	3	2	1	0	0	0	0	0	0
0	0	0	0	0	0	1	2	3	**4**	3	2	1	0	0	0	0	0	0
0	0	0	0	0	0	1	2	3	**4**	3	2	1	0	0	0	0	0	0

(b) 4-distance transformed image

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
4	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**
3	3	3	3	3	3	3	3	3	**4**	3	3	3	3	3	3	3	3	3
2	2	2	2	2	2	2	2	3	**4**	3	2	2	2	2	2	2	2	2
1	1	1	1	1	1	1	2	3	**4**	3	2	1	1	1	1	1	1	1
0	0	0	0	0	0	1	2	3	**4**	3	2	1	0	0	0	0	0	0
0	0	0	0	0	0	1	2	3	**4**	3	2	1	0	0	0	0	0	0
0	0	0	0	0	0	1	2	3	**4**	3	2	1	0	0	0	0	0	0

(c) 8-distance transformed image

習題 1.

請寫一個程式，並以下圖為原始影像進行 ***4-distance transform*** 和 ***8-distance transform***(下載miat400x200.raw)。

2. DT 影像的骨架抽取(Skeletonization)

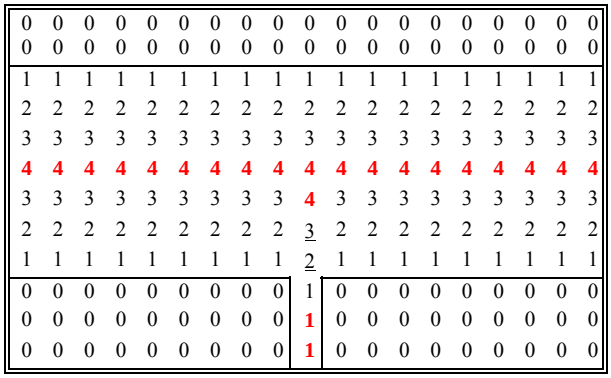
以 DT 來抽取而直影像的骨架包含兩個步驟：

Step 1. Distance transform

Step 2. Detection of the skeletal points

第一個步驟可以採用上一節所介紹的 4-distance 或 8-distance transform。至於第二個步驟，使用一個 local window 沿著左上右下的掃描順序偵測骨架的像素。對於每一像素，測試它的鄰近 8 個像素，如果它是區域最大值，也就是它的距離值大於或等於其週邊的所有像素，則可以視其為骨架點(skeletal point)。

這個方法可能造成骨架的不連續，如下圖：



一些影像後處理的技術可以用來解決或改善骨架不連續的情形，例如型態學影像處理的 openning 運算子，或是 Arcelli and Baja 演算法。

Arcelli-Baja 演算法

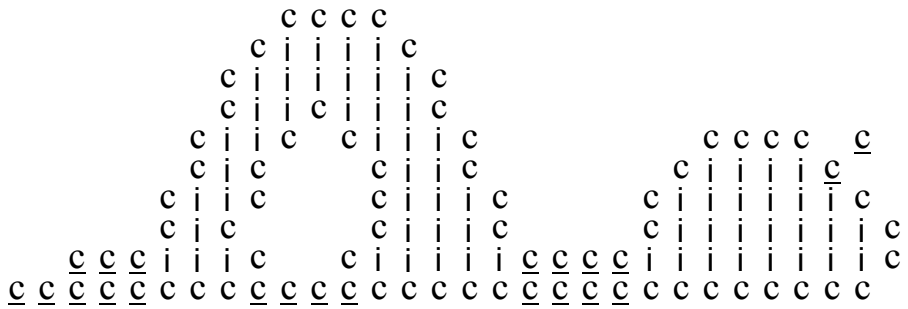
這個演算法應用在 4-distance 轉換後的影像，影像上的所有像素被分類為物件像素(A)和背景像素($\overline{A}$)，物件像素進一步再分成內部像素(*inner pixels, I*)和輪廓像素(*contour pixels, C*). 像素的距離值 d_4 如果為 1，則其屬於 C ，剩下的物件像素($d_4 \geq 2$)全屬於 I。

一個輪廓像素C，如果其相鄰 3x3 像素滿足以下兩個條件的其中一個條件，則定義該像素為***multiple*** C：

1. Neither the N-S (north-south), nor the W-E (west-east) neighbor pairs are such that one belongs to I and another to $\overline{A}$.
2. Any of the triples in the set { N-NE-E, E-SE,S, S-SW-W, W-NW-N } are such that the diagonal neighbor belongs to C, while the remaining two belong to

$\overline{A}$.

下圖經過這兩個條件的測試結果，C是偵測出來的***multiple***像素。

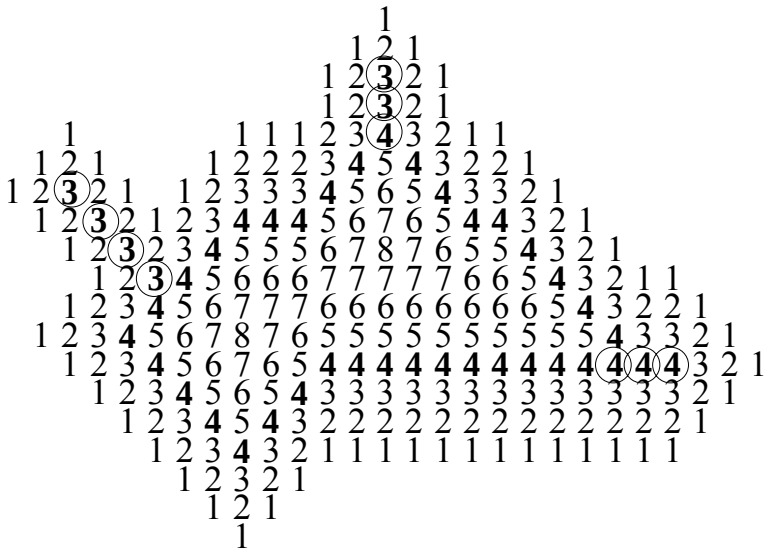


演算法：

步驟一、開始偵測輪廓像素(C)，測試每一個像素是否滿足兩個條件的其中一個，若是將其標示為骨架像素，其餘的輪廓像素被歸類至背景像素($\overline{A}$)。

步驟二、偵測第二層的物件像素(d_4=2)， 先將這些像素從 I 移至 C，再重複步驟一。

下圖是第三層和第四層的偵測結果：



疊加各層所偵測出的 ***multiple*** 像素，就得到了骨架影像。

習題 2.

應用 **Arcelli-Baja 演算法，抽取** miat400x200.raw 的骨架影像。

3.迭代刪除邊緣的細線化方法

以迭代方式一層一層刪除物件邊界的像素，最後保留中心的骨架(skeleton)，決定一個像素是否去除或保留係根據其與鄰近像素的關係。

P9	P2	P3

P8	P1	P4
P7	P6	P5

P1是為待處理之像素，而P2至P9 為其相鄰像素。物件像素值為1，背景像素值為0。

任何一個迭代刪除方式的細線化演算法必須滿足以下需求：

(a) 刪除的點不會消去端點；

(b) 刪除的點不會中斷連結；

(c) 不會引起區域的過度浸蝕。

Zhung and Suen(T.Y. Zhang and C.Y. Suen, “A fast parallel algorithm for thinning digital pattern,” Communications of the ACM, Vol. 27, No. 3, pp. 236-239,1984)提出一個平行偵測刪除邊緣的細線化方法，其演算法如下：

步驟(1)

沿左上右下順序對所有物件像素進行偵測，把同時滿足下列4條件的點加以標記：

(a) $2 \leqq N(P1) \leqq 6$

(b) $S(P1)=1$

(c1) $P2 \cdot P4 \cdot P6 = 0$

(d1) $P4 \cdot P6 \cdot P8 = 0$

其中N(P1)表示P1 的相鄰像素中，非0值的像素個數，即：

N(p1)=p2+p3+...+p9

S(*Pi*)則是依序從p2->p3->94….->p9當中，像素值從0變為1的次數。例如下圖

1	0	0
0	P1	1
0	1	1

可求出N(P1)=4，S(P1)=2。

條件(a)中，當N(P1)=0，表示P1 為一孤立點，當N(P1)=1，表P1 為端點，對孤立點或端點，不予刪除，否則物件會因細化而消失或導致線條退化。當N(P1)>6 時，則P1 為一內點，此情況也不予刪除，否則細化結果會產生hole。

條件(b)中，S(P1)為由P2,P3, …,P9，至P2 之序列中由”0”變”1”的次數，若S(P1)>1，則P1必為物件中某些部份(components)之一“橋樑”(bridge)，若將之刪除將會造成斷點，

(c1)及(d1)是用來刪除右邊及下邊的邊點及左上的轉角點。

步驟(2)、刪除標記的點

步驟(3)

沿左上右下順序對所有物件像素進行偵測，把同時滿足下列4條件的點加以

標記：

(a) $2 \leqq N(P1) \leqq 6$

(b) $S(P1)=1$

(c2) $P2 \cdot P4 \cdot P8 = 0$

(d2) $P2 \cdot P6 \cdot P8 = 0$

(a)和(b)與步驟(1)相同，(c2)及(d2) 則用來刪除左邊及上邊的邊點及右下的轉角點。

步驟(4) 刪除標記的點。

步驟(5) 重複上述步驟(1)到(4)直到沒有標記的點為止。

4. 細線化程式碼

```
//-----------------------------------------------------------------
// thinning.h
// thinning of binary image
// MIAT Lab, Kaohsiung, Taiwan
// Algorithm of Zhung and Suen(1984)
// revised by CHEN Ching-Han, 2004.11.09
//-----------------------------------------------------------------
bool thinning(uc2D &ima)
{
  bool success = false ;
  bool success2 = false;
  int count ;
  int count2;
  int num_transitions ;
  int transitions ;
  uc2D temp_array,temp;
  temp_array.Initialize(ima.nr,ima.nc);
  temp.Initialize(ima.nr,ima.nc);
  for (int i = 1; i < ima.nr-1; i++) {
    for (int j = 1; j < ima.nc-1; j++) {
      count = 0;
      num_transitions = 0;

      // check  for N(p)
      if (ima.m[i][j] == 255) {
```

```
if (ima.m[i-1][j-1] != 0)
  count++;
if (ima.m[i][j-1] != 0)
  count++;
if (ima.m[i+1][j-1] != 0)
  count++;
if (ima.m[i+1][j] != 0)
  count++;
if (ima.m[i-1][j] != 0)
  count++;
if (ima.m[i+1][j+1] != 0)
  count++;
if (ima.m[i][j+1] != 0)
  count++;
if (ima.m[i-1][j+1] != 0)
  count++;

if (count != 8) {
  // 2 <= N(p) <= 6
  if (count >= 2 && count <= 6) {
    if(ima.m[i-1][j] == 0 && ima.m[i-1][j+1] == 255)
      num_transitions++ ;
    if(ima.m[i-1][j+1] == 0 && ima.m[i][j+1] == 255)
      num_transitions++ ;
    if(ima.m[i][j+1] == 0 && ima.m[i+1][j+1] == 255)
      num_transitions++ ;
    if(ima.m[i+1][j+1] == 0 && ima.m[i+1][j] == 255)
      num_transitions++ ;
    if(ima.m[i+1][j] == 0 && ima.m[i+1][j-1] == 255)
      num_transitions++ ;
    if(ima.m[i+1][j-1] == 0 && ima.m[i][j-1] == 255)
      num_transitions++ ;
    if(ima.m[i][j-1] == 0 && ima.m[i-1][j-1] == 255)
      num_transitions++ ;
    if(ima.m[i-1][j-1] == 0 && ima.m[i-1][j] == 255)
      num_transitions++ ;

    //S(p) = 1
```

```
            if (num_transitions == 1) {
               // if p2 * p4 * p6 = 0
               if (ima.m[i-1][j] == 0 || ima.m[i][j+1] == 0 ||
                   ima.m[i+1][j] == 0){
                  // if p4 * p6 * p8 = 0
                  if(ima.m[i][j+1] == 0 || ima.m [i+1][j] == 0 ||
                     ima.m[i][j-1] == 0) {
                     temp_array.m[i][j] = 0;
                     success = true;
                  }
                  else
                     temp_array.m[i][j] = 255;
               }
               else
                  temp_array.m[i][j] = 255;
            }
            else
               temp_array.m[i][j] = 255 ;
         }
         else
            temp_array.m[i][j] = 255 ;
      }
      else
         temp_array.m[i][j] = 255;
   }
   else
      temp_array.m[i][j] = 0;
  }
}
//copy thinned image back to original
for (int a = 0; a < ima.nr; a++) {
  for (int b = 0; b < ima.nc; b++)
     ima.m[a][b] = temp_array.m[a][b];
}

// step 2 of the thinning algorithm
for (int k = 0; k < ima.nr; k++) {
  for (int l = 0; l < ima.nc; l++) {
```

```
count2 = 0;
transitions = 0;
if (ima.m[k][l] == 255) {
   if (ima.m[k-1][l-1] != 0)
      count2++;
   if (ima.m[k][l-1] != 0)
      count2++;
   if (ima.m[k+1][l-1] != 0)
      count2++;
   if (ima.m[k+1][l] != 0)
      count2++;
   if (ima.m[k-1][l] != 0)
      count2++;
   if (ima.m[k+1][l+1] != 0)
      count2++;
   if (ima.m[k][l+1] != 0)
      count2++;
   if (ima.m[k-1][l+1] != 0)
      count2++;

   if (count2 != 8) {
      if (count2 >= 2 && count2 <= 6) {
         if(ima.m[k-1][l] == 0 && ima.m[k-1][l+1] == 255)
            transitions++ ;
         if(ima.m[k-1][l+1] == 0 && ima.m[k][l+1] == 255)
            transitions++ ;
         if(ima.m[k][l+1] == 0 && ima.m[k+1][l+1] == 255)
            transitions++ ;
         if(ima.m[k+1][l+1] == 0 && ima.m[k+1][l] == 255)
            transitions++ ;
         if(ima.m[k+1][l] == 0 && ima.m[k+1][l-1] == 255)
            transitions++ ;
         if(ima.m[k+1][l-1] == 0 && ima.m[k][l-1] == 255)
            transitions++ ;
         if(ima.m[k][l-1] == 0 && ima.m[k-1][l-1] == 255)
            transitions++ ;
         if(ima.m[k-1][l-1] == 0 && ima.m[k-1][l] == 255)
            transitions++ ;
```

```
                if (transitions == 1) {
                   // if p2 * p4 * p8 = 0
                   if(ima.m[k-1][l] == 0 || ima.m[k][l+1] == 0 ||
                      ima.m[k][l-1] == 0){
                      // if p2 * p6 * p8
                      if(ima.m[k-1][l] == 0 || ima.m[k+1][l] == 0 ||
                         ima.m[k][l-1] == 0){
                          temp.m[k][l] = 0;
                          success2 = true;
                      }
                      else
                         temp.m[k][l] = 255;
                   }
                   else
                      temp.m[k][l] = 255;
                }
                else
                   temp.m[k][l] = 255;
             }
             else
                temp.m[k][l] = 255;
          }
          else
             temp.m[k][l] = 255;
       }
       else
          temp.m[k][l] = 0;
    }
  }

  for (int a = 0; a < ima.nr; a++) {
     for (int b = 0; b < ima.nc; b++)
       ima.m[a][b] = temp.m[a][b];
  }
  if(success || success2)   return true;
  else return false;
}
```

使用範例：

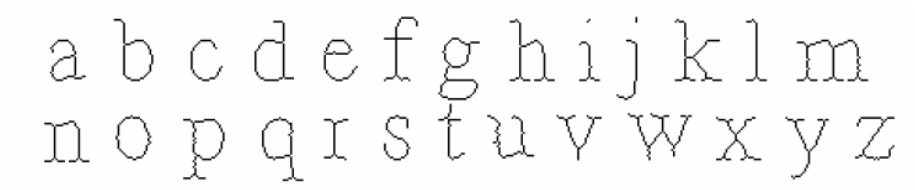
```
void main()
{
   uc2D ima;
   int nr=120,nc=120;
   ifstream in("bin.raw",ios::binary);
   ofstream out("thin.raw");
   ima.Initialize(nr,nc);
   for(int i=0;i<ima.nr;i++)for(int j=0;j<ima.nc;j++)
      ima.m[i][j]=in.get();
   bool continue;
   while(1)
   {
     continue = thin(ima);
     if(continue==0)break;
   }
   for(int i=0;i<ima.nr;i++)for(int j=0;j<ima.nc;j++)out<<ima.m[i][j];
}
```

習題 3.

使用上述程式抽取 miat400x200.raw 的骨架影像，並與習題 2 所得到的影像重疊，找出兩個結果的差異，並加以說明。

6. (option)細線化結果的改善

細線化容易產生骨架毛邊和交接點的骨架變形問題，思考如何將以後處理來改善此一結果。


a b c d e f g h i j k l m
n o p q r s t u v w x y z