
Subject: Re: Thinning algorithm without for loops
Posted by [nathan12343](#) on Tue, 07 Aug 2007 21:57:05 GMT
[View Forum Message](#) <> [Reply to Message](#)

On Aug 7, 12:45 pm, nathan12343 <nathan12...@gmail.com> wrote:
> On Aug 7, 7:28 am, Conor <cmanc...@gmail.com> wrote:
>
>
>
>> On Aug 6, 3:55 pm, "Jeff N." <jnett...@utk.edu> wrote:
>
>>> On Aug 6, 3:31 pm, nathan12343 <nathan12...@gmail.com> wrote:
>
>>>> Hi all-
>
>>>> I'm trying to impliment the Zhang-Suen thinning algorithm in IDL.
>>>> This particular algorithm decides whether a pixel needs to be deleted
>>>> or not based on properties of the pixels immediately surrounding the
>>>> pixel we are concerned with (i.e. a pixel's 8-neighbors). This
>>>> naturally lends its self to for loops. Let's say I have an image, a
>>>> 512X512 array of bytes. The code iteratively scans over each pixel
>>>> and determines whether it needs to be set to 0 based on the Zhang-Suen
>>>> thinning rules. What I can't figure out is how to scan the images
>>>> without for loops. If I use for loops I can easily index the pixels
>>>> immediately surrounding image[i,j] by saying image[i-1,j] or image[i
>>>> +1,j-1], etc.
>
>>>> Does anyone know of a way to do this kind of indexing in an image
>>>> without the use of for loops?
>
>>>> -Nathan Goldbaum
>
>>> I've never done this myself, so someone else will probably have to
>>> give you details if you need more help, but I think the function you
>>> need is the SHIFT() function. Have a look at the help files for that
>>> and see what you think.
>
>>> Jeff
>
>> It really depends on just what the thinning algorithm needs to check.
>> Can you be specific about that? In general though, the shift function
>> is probably the way to go, although it will be rather ugly at the same
>> time (since you'll have to call it 8 times). Let's pretend for a
>> moment that your thinning algorithm is very simple: namely, let's
>> imagine that you want to ignore all pixels where the average
>> surrounding pixels have a value greater than some threshold.
>
>> ; make a fake image

```

>> img = byte( randomu(seed,512,512)*100 )
>> ; array to hold the total values
>> tot = intarr( 514,514 )
>
>> ; pad img with zeroes on all sides, because shift() wraps around an
>> array and you don't want values from the other side
>> img = [[fltarr(514)],[fltarr(1,512),img,fltarr(1,512)],[fltarr(514 )]]
>
>> ; now find the total value of all neighboring pixels
>> tot += shift(img,1,-1)
>> tot += shift(img,1,0)
>> tot += shift(img,1,1)
>> tot += shift(img,0,-1)
>> tot += shift(img,0,1)
>> tot += shift(img,-1,-1)
>> tot += shift(img,-1,0)
>> tot += shift(img,-1,1)
>
>> ; now find the average
>> avg /= 8
>
>> ; now pull the original image and the totals out of the padded arrays
>> img = img[1:512,1:512]
>> avg = avg[1:512,1:512]
>
>> ; finally, select everything below a certain threshold:
>> w = where( avg lt threshold )
>
>> It's ugly, and it probably isn't the best solution, but it will get
>> the job done and it will probably be much faster than a loop
>> solution. Of course, that depends on just what the thinning algorithm
>> does, and whether or not it can be generalized in such a fashion.
>
> I don't know if this particular algorithm can be generalized like
> you're doing with the Shift function. If the pixels surrounding the
> pixel of interest are numbered like so:
>
>           P[8] P[1] P[2]
>
>           P[7] P   P[3]
>
>           P[6] P[5] P[4]
>
> A pixel is flagged to be deleted if it is part of the foreground (i.e.
> has a value of 1) and one of its 8-neighbors is part of the background
> and, in a first iteration:
>
> 1. The sum of the numbered pixels is greater than or equal to 2 and

```

- > less than or equal to 6.
- > 2. The number of 0-1 transitions in the ordered sequence
- > $P[1], P[2], \dots, P[8], P[1]$ is exactly 1
- > 3. If $P[3]*P[5]*P[7]=0$
- > 4. If $P[1]*P[3]*P[5]=0$
- >
- > And in the second iteration:
- >
- > The same two original conditions and the additional conditions:
- >
- > 5. If $P[1]*P[5]*P[7]=0$
- > 6. If $P[1]*P[3]*P[7]=0$
- >
- > And that's it. I see now how to implement all of these conditions
- > with the shift function, but what about condition number 2? Either
- > way, I'm pretty sure moving most of this stuff out of the for loop
- > will make the code run much quicker, thanks for your help!

Thanks for your help, Conor, the shift function appears to have done the trick.

This code implements the first iteration of Zhang-Suen thinning without a single for loop!

```
PRO zsthin,img,thinimg
```

```
siz=size(img)
```

```
;Array to hold the sums we're looking for
tot=lonarr(siz[1],siz[2])
```

```
tot+=shift(img,1,0)
tot+=shift(img,1,1)
tot+=shift(img,1,-1)
tot+=shift(img,0,1)
tot+=shift(img,0,-1)
tot+=shift(img,-1,0)
tot+=shift(img,-1,1)
tot+=shift(img,-1,-1)
```

```
cond3=intarr(siz[1],siz[2])
```

```
cond3[:,*]=1
```

```
cond3*=shift(img,1,0)
cond3*=shift(img,-1,0)
cond3*=shift(img,0,-1)
```

;4. If $P[1]*P[3]*P[5]=0$

```
cond4=intarr(siz[1],siz[2])
cond4[*,*]=1
```

```
cond4*=shift(img,0,1)
cond4*=shift(img,1,0)
cond4*=shift(img,0,-1)
```

;2. The number of 0-1 transitions in the ordered sequence
;P[1],P[2],...,P[8],P[1] is exactly 1

```
p1=shift(img,0,1)
p2=shift(img,1,1)
p3=shift(img,1,0)
p4=shift(img,1,-1)
p5=shift(img,0,-1)
p6=shift(img,-1,-1)
p7=shift(img,-1,0)
p8=shift(img,-1,1)
```

```
cond2=intarr(siz[1],siz[2])
```

```
p=[[[p1]],[[p2]],[[p3]],[[p4]],[[p5]],[[p6]],[[p7]],[[p8]],[[p1]]]
```

FOR i=0,7 DO BEGIN

```
  wh=where(p[*,* ,i] eq 0 AND p[*,* ,i+1] eq 1)
```

```
  cond2[wh]+=1
```

ENDFOR

```
tvsc1,cond2
```

```
wh=where(cond2 eq 1)
```

```
cond2[*,*]=0
```

```
cond2[wh]=1
```

```
wh=where(tot GE 2 AND tot LE 6 AND cond3 eq 0 AND cond4 eq 0 AND cond2
eq 1)
```

```
wh11=intarr(siz[1],siz[2])
```

```
wh11[wh]=1
```

```
newimg=img-wh11
```

newimg>=0

END

I'll do the second subiteration in a bit, should be too hard.
