Algorithms Rule Supreme

Lloyd Moore

FS Studio

UBM

#ESCcon
f

Algorithms Rule Supreme

Lloyd Moore, Senior Embedded Systems Engineer
30 years of embedded and machine vision software experience

> "**Code optimizations may get 2-4x improvement**
> **Algorithm changes can get more than 10x**"

We are going to look at how to tailor an algorithm to best fit the problem definition and improve performance

Agenda

Definition of connectivity

/blob analysis

Algorithm analysis

“Traditional”

approach

Algorithm analysis

“Wanderer” approach

Algorithm analysis

“Single pass” approach

Comparison and Summary

Comparing Algorithms

Assume images are 100 x 100 pixels – simple math

Algorithms will be pseudo-code and we'll count operations

Algorithms somewhat simplified for presentation

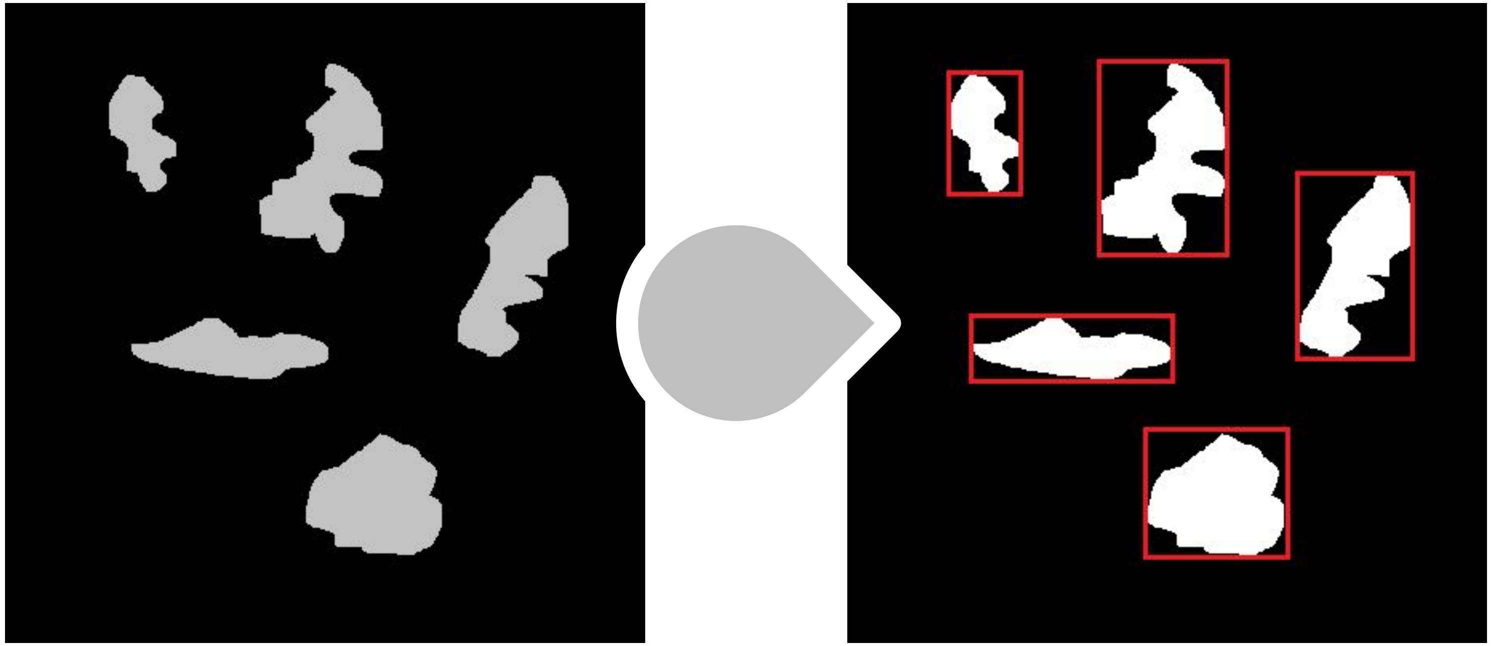
Will consider memory/cache friendliness separate from operations

Will consider advantages & disadvantages for image content

Consider **3 algorithm solutions:**

1. "Traditional" Algorithm
2. "Wanderer" Algorithm
3. "Single Pass" Algorithm

What is Connectivity Analysis



Also called **Blob Analysis**
Goal is to determine which pixels in an image are adjacent
Transforms a group of individual pixels into one “object”
For our discussion we will record a bounding box for the “object”

“Traditional” Algorithm - Problem

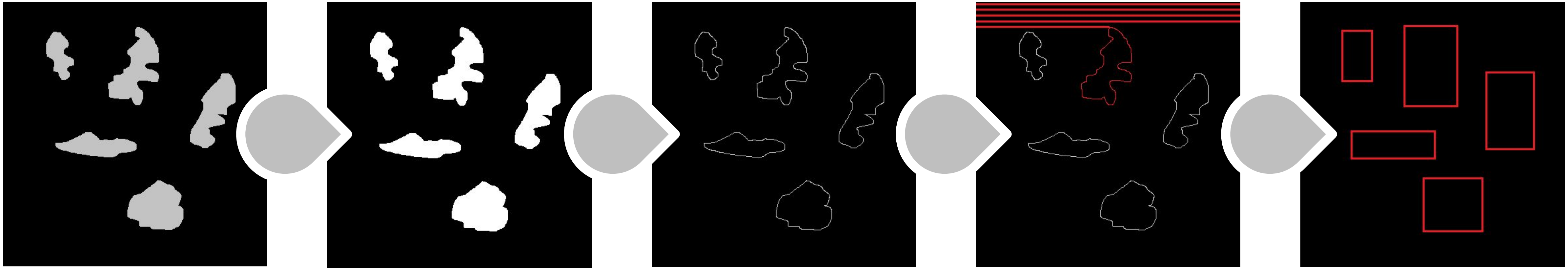
Will use a **“text book”** approach to start

Won’t make any assumptions about blobs in image

Serves as a **baseline** for other algorithms

Likely **wouldn’t** want to **use this in real life**

"Traditional" Algorithm - Outline



1 Threshold Image

2 Apply edge detection kernel

3 "Walk" image to find start of an outline

4 Follow the outline to find the blob, and update bounding box

5 Continue until all blobs found, end of the image

“Traditional” Algorithm - Complexity

Threshold:

For each pixel in image

If value > Threshold then
Value = 255
Else
Value = 0

Edge Detection Kernel:
Assume: 3x3 kernel, stride of 1, kernel stays inside image
For each image position Ix

For each image position Iy

value = 0
For each kernel position Kx

For each kernel position Ky

value += image[Ix, Iy] * kernel[Kx, Ky]
Target[Ix, Iy] = value

els = 10000 loops
pare + test = 4 operations
are + test + write = 4 operations

30,000 Ops

loops
pare + test = 4 operations
loops
pare + test = 4 operations

pare + test = 4 operations

pare + test = 4 operations
+ 3 multiply + 3 add + 1 write = 13 operations
ltiply + 1 add + 1 write = 5 operations

3 + 1 + 4) * 98 + 4) * 98 = 682,276 Ops

“Traditional” Algorithm - Complexity

Walk Image, find outline start:
For each pixel in image

If value == 255 then
Trace outline();

Trace blob outline:
For each pixel in border:
If Image[x-1, y] == 255 then x-- (for 5 cases)

If x < min_x then min_x = x (for 4 cases)

If x = starting_x && y == starting_y then break
Image[x, y] = 0 (erase current pixel)

= 10000 loops
+ test = 4 operations
= 3 operations
ore call overhead

ps

read + multiply + add + compare + test) = 25 operations
d + add/sub + write = 3 operations
+ compare + test) = 16 operations
: write = 0.25 operations
compare + and + test = 8 operations
nultiply + add + write = 5 operations

= 57.25 Ops per border pixel

0000 = 832,276 operations per image
er border pixel of all blobs

“Traditional” Algorithm – Strengths/Weakness

Strengths

Simple to **implement**

Simple to **understand**

Mostly **independent** of image content

Weakness

Pretty **slow**

Multiple **passes** over image

Multiple **working** images

Not **cache** friendly

Original image **destroyed**

“Wanderer” Algorithm - Problem

- Images consist of **50 to 200 very thin blobs**
- Imaging **environment is controlled**
- **No “large” blobs**
- Example comes from a **real world optimization project**
- Blobs were actually **fibers of a material**

NOTE: *Example images will show on a few blobs*

“Wanderer” Algorithm - Outline

1. Threshold Image
2. Find Blob Start
3. Explore Blob, Updating Bounding Box
4. Double Check Blob Fully Explored
5. Continue Until End of Image Reached

“Wanderer” Algorithm - Complexity

Threshold: Same as “Traditional” **80,000 Ops**
Find Blob Start: Same as “Traditional” **70,000 Ops**

Explore Blob:
For each pixel in blob:
Explore adjacent the 8 adjacent pixels

```
explorer_pointer = cur_blob_start + fixed_offset      read + write + add = 3 operations
if(*explorer_pointer == untouched_pixel)              dereference + read + test = 3 operations
    accumulate_pixel(this_x, this_y)                  Assume 75% of the time: 0.75 *(2 push + call) = 2.25 operations
*explorer = *explorer & constant_tag                  dereference + read + and + write = 4 operations

accumulate_pixel:
    If x < min_x then min_x = x (for 4 cases)         4 * (2 read + compare + test) = 16 operations
                                                      Assume 25% trigger if clause: write = 0.25 operations
    ++pixel_count                                     read + increment + write = 3 operations
    return                                            return = 1 operation
```

3 + 3 + 2.25 + 4 + 0.75 * (16 + 0.25 + 3 + 1) =
27.5 operations per blob pixel (approximately actual is 27.4375)

“Wanderer” Algorithm - Complexity

Explore Blob:

For each pixel in blob:

Move to next pixel: explore right, down, down & right, down & left: 4 cases assume 50% hit so count 2 cases

explorer_pointer = explorer_pointer + fixed_offset	read + write + add = 3 operations
if(*explorer_pointer > completed_pixel	dereference + read + subtract + test = 4 operations
&& *explorer_pointer < untouched_pixel)	and + subtract + test = 3 operations
cur_blob_start = explorer_pointer	write = 1 operation
cur_coordinate += const_offset	read + add + write = 3 operations

2 * (3 + 4 + 3 + 1 + 3) = 28 operations per blob pixel

Double Check Blob Fully Explored:

For each pixel current blob bounding box:

if(*explorer_pointer > completed_pixel	dereference + read + subtract + test = 4 operations
&& *explorer_pointer < untouched_pixel)	and + subtract + test = 3 operations
cur_blob_start = explorer_pointer	write = 1 operation
cur_blob_x = cur_offset % width	2 reads + modulus + write = 4 operations
cur_blob_y = cur_offset / width	2 reads + divide + write = 4 operations

4 + 3 + 1 + 4 + 4 = 16 operations per current blob counted pixel

"Wanderer" Algorithm - Complexity

Threshold:	Same as "Traditional"	**80 000 Ops**
Find Blob Start:	Same as "Traditional"	**0,000 Ops**
Explore Blob:	27.5 + 28 ops	**55.5 Ops per Blob Pixel**
Double Check:	16 ops, assume executes 3x	**8 Ops per Blob Pixel**

Total = 80,000 + 70,000 = 150,000 operations per image
+ 55.5 + 48 = + 103.5 operations per blob pixel

Comparison and Summary

Algorithm:
Traditional Pixel Wanderer

Ops per Image:
832,276
150,000

+

Ops per Feature:
57.25 per blob border
103.5 per blob area pixel

<u>OBSERVATIONS:</u>

Performance **GREATLY** depends on image contents
Wanderer faster for empty image, **worse** for large blobs
This application had long thin blobs, most only 1 or 2 pixels in width;
blob area approximated blob border pixels in practice

“Wanderer” Algorithm – Strengths/Weakness

Strengths

15 to 30x **faster** for target image content vs commercial library

Single **copy** of image

Image **altered** but available

Weakness

Complex to implement

HIGHLY dependent on image content

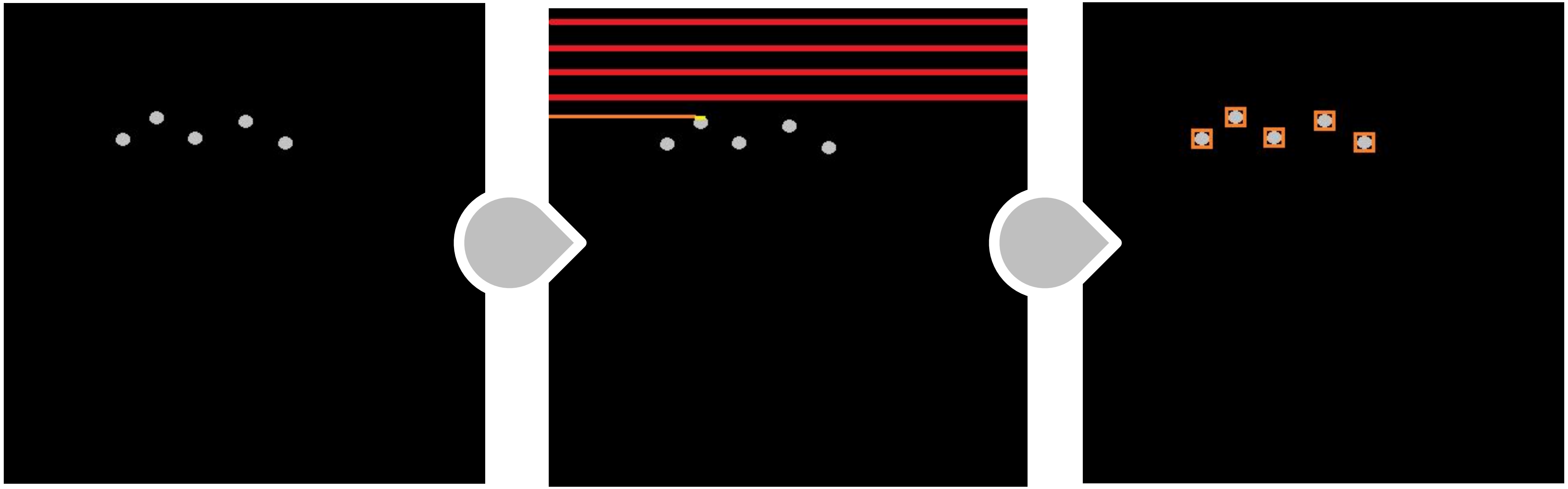
Multiple passes over image

Not **cache** friendly

"Single Pass" Algorithm - Problem

- Track 5 to 10 small round objects per frame
- Run on VERY small processors, including micro-controllers
- Target processor need not hold full video frame, only current pixel
- Example comes from a real world project
- Image content mostly controlled via narrow band optical filter

“Single Pass” Algorithm - Outline



1 For each pixel in the image

2 Threshold the pixel and detect segment start and ends

3 When a segment is complete add it to the connecting blob structure

"Single Pass" Algorithm - Complexity

Setup variables:
forming_vector = false
pixel_scanner = image_start
current_x = current_y = 0;

Insert blob line:
For each blob
if this_y == last_y + 1
if min_x >= blob_min_x or
max_x <= blob_max_x
blob_last_y = this_y
blob_min_x = min_x
blob_max_x = max_x
if min_x < box_min_x
box_min_x = min_x
if max_x > box_max_x
box_max_x = max_x

eration
write = 2 operations
operations

1 + 2 + 2 = 5 operations per image

at all times (worst case)
compare, test = 5 operations
pare, test, or = 5 operations
, compare, test = 4 operations
perations <=Only for 1 blob
= 2 operations <=Only for 1 blob
= 2 operations <=Only for 1 blob
pare, test = 4 operations <=Only for 1 blob
= 2 operations <=Only for 1 blob, 50%
pare, test = 4 operations <=Only for 1 blob
= 2 operations <=Only for 1 blob, 50%

2 + 2 + 4 + 0.5 + 4 + 0.5) = 155 operations per blob line

“Single Pass” Algorithm - Complexity

Walk the image:

```
For each pixel in the image:                  100 x 100 pixels = 10000 loops
  if *pixel_scanner > threshold                   dereference + read + compare + test = 4 operations
    Assume: 1% hit image is mostly black
      if not forming_vector                       read + compare + test = 3 operations  <=Take worst case
            starting_x = max_x = current_x        read + 2 write = 3 operations
            starting_y = current_y                read + write = 2 operations
            forming_vector = true                 write = 1 operation
      else
           max_x = current_x                  read + write = 2 operations   <=Not worst case
    else
           if forming_vector                      read + compare + test = 3 operations   <=Not worst case
         insert_blob_line()                   from previous slide  <=Counted per blob line
         forming_vector = false               write = 1 operation      <=Not worst case
    ++pixel_scanner; ++current_x; ++current_y          3* (read + increment + write) = 9 operations
    if current_x > image_width                read + compare + test = 4 operations
         if forming_vector insert_blob_line()          read + compare + test = 3 operations   <=Image Row
         ++current_y; current_x = 0;              read + increment + 2 write = 4 operations  <=Image Row
         forming_vector = false               write = 1 operation       <=Image Row
```

10000 * 4 + 0.01 * (3 + 3 + 2 + 1) + (9 + 4) + 10 * (3 + 4 + 1) = 40093.09 => 40,093 + 5 setup =
40,098 ops per image + 155 ops per blob line

“Single Pass” Algorithm – Strengths/Weakness

Strengths

Extremely **fast**, though no direct benchmark

Single pass through image, and only need to have one pixel of the image at any time

Original image untouched

Simple to implement

Very cache **friendly**

Weakness

Performance **suffers** with large number of blobs

Have to deal with combining blob fragments in some cases; did not address that here as didn’t need it for this particular case

Comparison and Summary

Algorithm:
Traditional
Wonderer
Single Pass

Ops per Image:
832,276
150,000
40,098

Ops per Feature:
57.25 per blob border pixel
103.5 per blob pixel
155 per blob line

OBSERVATIONS:

Note ops per image constantly goes down, consider an empty image
Most blob lines will have many pixels so 155 ops per line isn't that bad
Consider a completely white image: single pass still better
Actual implementation also had noise filter to consolidate blob lines

Final Thoughts

- Matching the algorithm to the expected use case and input can greatly improve performance
- These gains are complimentary and additive to other optimization techniques
- Consider radically different approaches – "Single Pass" cannot be clearly evolved from "Traditional" or "Wanderer" algorithms
- ALWAYS measure actual performance and use a wide variety of input

Speaker/Author Details

Http://FSStudio.com
LloydMoore@FSStudio.com

FS Studio

Thank You!

Questions?

UBM esc

@ESC_Conf
#ESCcon