Algorithms in C#: Connected Component Labeling

8 May 2008 2:57 AM | **11**

I have to admit, I'm not very good at interviews. For some reason my mind isn't trained well for sorting linked lists or balancing binary trees on the whiteboard. I have no problem designing 600+ type hierarchies and building complex object-oriented frameworks, but typical interview questions were never an area where I was shining. Maybe it's because I like to think slowly and take my time, maybe for some other reasons :) Anyway, just out of the blue I decided to implement an algorithm in C# with the goal to improve my algorithm skills and to enjoy the power and expressiveness of my most favorite language and platform.

As it usually goes, I first sat down and spent three hours coding the whole thing, and *after* that I did some online research, which revealed the algorithms official name:

Connected component labeling

The goal is, given a 2D-matrix of colors, to find all adjacent areas of the same color, similar to the flood-fill algorithm used in Paint.

Standard (existing) solution - Union-Find

Online research had revealed that there is a standard algorithm for this that heavily employs **Union-Find**. The union-find approach starts with a disjoint matrix, where every element is its own set and then iteratively joins (unions) neighboring sets together. The secret of Union-Find is twofold:

- Extremely fast (constant-time) union of two sets
- Extremely fast (almost constant-time) determining whether two points belong to the same set or not

I remember when I was at the university, I even knew how to prove that the complexity of Union-Find is bounded by the inverse Ackermann function, which, believe me, *grows very slowly*. The important thing that I carried out for the future is that for all practical purposes you can safely consider this O(1).

My solution

I was somewhat proud to have implemented a different approach. If you have seen this anywhere before, I wouldn't be surprised though, so please bear with me. The idea is to split the "image" into horizontal stripes ("spans") of the same color. Then scan all the rows of the image row-by-row, top-to-bottom, and for each row, for each span in a row, attach the span to the neighbor spans of the previous row. If you're "touching" more than one existing component, join them into one. If you're not touching any pixels of the same color, create a new component. This way, we split all horizontal spans into three generations: 0, 1 and 2 (like the .NET Garbage Collector!). Generation 0 is the current row, Generation 1 is the row above it, and Generation 2 are all the rows above these two. As we finish processing a row, Generation 1 is added to Generation 2 and Generation 0 becomes Generation 1.

MSDN Code Gallery

I published the source code at **http://code.msdn.microsoft.com/ConnectedComponents**. MSDN Code Gallery is an awesome new website where you can publish your code samples and source code. I discovered that I'm not the only one to publish algorithms in C#: see for example, **A* path finding**.

Performance challenge

I'd be very curious to measure the performance of my current code. I know it isn't optimal because there is at least one place where I can do considerably better. For the profiling wizards among you, I won't reveal where it is for now. For the tuning wizards, if my algorithm takes 100% time, how many percent can you get? I myself expect (but not guarantee) to be around 5-10% faster. Can anyone do better?

Also, I'd be curious how my algorithm performs compared to the classical union-find implementation. Anyone interested to implement and see if it does better? (I hope I won't regret posting this when someone shares an implementation which is 100 times faster than mine...)

Cleaner code challenge

I dare to hope that my code is more or less clean and more or less well-factored. However I learned one lesson in life - whenever I start thinking that I'm good, life immediately proves otherwise. That's why I welcome any improvement suggestions. I understand that on such Mickey Mouse-size projects it doesn't really matter that much if the code is clean or not, but anyway. The design could be better, the use of .NET and C# could be better, the distribution of responsibilities could be better, my OOP kung-fu could be better. I'd like to use little projects like this to learn to code better.

Language challenge

Care to implement it in **F#** or, say, **Nemerle**? This is your chance to show the world that C# is not the only .NET language! Of course, non-.NET languages are highly welcome as well. I'd personally be interested in, for example, Haskell, Lisp, Prolog, Scala, Nemerle, Boo, (Iron)Ruby, (Iron)Python, Smalltalk, Comega, SpecSharp, SML.NET and so on.

Silverlight challenge

Have you installed Silverlight 2 beta? Need an idea what to implement? ;-)

Comments

Kirill Osenkov - MSFT
13 May 2008 1:27 AM

Brian, a developer on the F# team, has posted the F# solution which uses the Union-Find algorithm:

http://lorgonblog.spaces.live.com/blog/cns!701679AD17B6D310!220.entry

It's much faster than my algorithm :) And looks shorter. Highly recommended.

Morgan Cheng
21 May 2008 6:16 AM

I happens to investigate this algorithm recently.

I have to adimit that the track row instead of point is a ingenious idea, but there is a lot collections are created and then released. So, I tried to implement it in other ways to validate the performance.

The result is interesting:

For most of the time the pixel-by-pixel algorithm is much faster than span-by-span algorithm. Pixel-by-pixel is not defeated when color blob is sparse. That is, "White percentage" bar is to very left or very right.

Kirill Osenkov - MSFT
21 May 2008 9:00 PM

Interesting! Thanks Morgan. I'll have to implement the classic pixel-by-pixel algorithm next time to measure the performance difference.

Morgan Cheng
21 May 2008 10:35 PM

I'm in rush yesterday and have no time to analyze the reason. Now, I believe that the reason is that the randomly generated graph is not appropriate for span-by-span detection. Since it is random, black and white pixels are even distributed. So, there is not long span there; each span has only few pixels. In worst case, each span has only one pixel. As a result, the power of span is not utilized.

I use the two different way on real image. It turns out that span-by-span way is better than pixel-by-pixel way, since in real photo, same color is generally in blob. In average, the time cost ratio is 3:1. So, the span-by-span algorithm is really helpful! thanks.

The 3-generation way like .net GC is really brilliant idea! But I implement span-by-span in traditional way. Each span is assigned a blobID . For each span, if there is no intersection with up row, a new blob id is assigned; if there is only one intersection, the up span blob ID is reused; if there is more than one intersection, use the smallest blob id, and a boundTo mapping of id-to-id is updated. After the parsing, we can get the directBoundTo from boundTo, then each blob has a unique blob ID.

Below is just some sample code. The performance is better than 3-gen way. Perhaps mainly due to less function call and collection used.

```
var boundTo = new Dictionary<int, int> ();

int blobID = 0;

for (int y = 0; y < height; ++y)

{

   var rowSet = new StrikeList<Color>();

   allStrikes.Add(rowSet);

   {

      Strike<Color> strike = null;

      Color prevPixel = Color.Empty;

      for (int x = 0; x < width; ++x)

      {

         Color currPixel = field[x, y];

         if (currPixel != prevPixel)

         {

            if (strike != null)

            {

               strike.EndX = x - 1;

               rowSet.Add(strike);

               strike = null;

            }

            prevPixel = currPixel;

            if (currPixel != Color.Empty)

            {

               strike = new Strike<Color> { StartX = x, Color = currPixel, Y = y };

            }

         }

      }

      if (strike != null)

      {

         strike.EndX = width - 1;

         rowSet.Add(strike);

      }

   }
```

```
        StrikeList<Color> prevRowSet = (y > 0) ? allStrikes[y - 1] : new StrikeList<Color>();
        foreach (var strike in rowSet)
        {
            List<Strike<Color>> insected = prevRowSet.Where(s => s.IntersectWith(strike)).ToList();
            if (insected.Count == 0)
            {
                strike.BlobID = (++blobID);
                boundTo[blobID] = blobID;
            }
            else if (insected.Count == 1)
            {
                strike.BlobID = insected[0].BlobID;
            }
            else
            {
                var minBlob = insected[0];
                for (int i = 0; i < insected.Count; ++i)
                {
                    if (insected[i].BlobID < minBlob.BlobID)
                    {
                        minBlob = insected[i];
                    }
                }
                strike.BlobID = minBlob.BlobID;
                foreach (var s in insected)
                {
                    if (s != minBlob)
                    {
                        boundTo[s.BlobID] = minBlob.BlobID;
                    }
                }
            }
        }
    }
    var directBoundTo = new Dictionary<int, int>();
    foreach (var key in boundTo.Keys)
```

```
{
   int target = boundTo[key];
   while (boundTo[target] != target)
   {
      target = boundTo[target];
   }
   directBoundTo[key] = target;
}
```

Kirill Osenkov - MSFT
17 Jun 2008 6:17 PM

Morgan, thanks so much for sharing this, this is really interesting. It's great that you've found a way to improve performance and I think you're right - span tracking isn't really shining on random images.

I like your idea, it saves up memory because you don't have to initialize throw away collections every time.

Balint
11 Mar 2010 12:55 PM

Could you give an other link where I can download your c# solution for the labeling? The MSDN link does not works (at leas for me)

Thanks

Kirill Osenkov - MSFT
11 Mar 2010 12:59 PM

Balint - it works fine for me, but just in case, here's another link for you:

http://dl.dropbox.com/u/3737073/ConnectedComponents.zip

Let me know if this works.

Algorithm
5 Apr 2010 11:49 PM

who create the random algorithm in c#

kiran
16 Apr 2010 4:57 AM

i need this one.because to use this one for my project.

thank u

Ugly
30 Nov 2010 7:22 PM

There is a much much faster way .. www.iis.sinica.edu.tw/papers/fchang/1362-F.pdf or google "A Linear-Time Component-Labeling Algorithm Using Contour Tracing Technique" by Fu Chang, Chun-Jen Chen, and Chi-Jen Lu

It also has the advantage it extracts (traces) the contour as it does connected component labelling and you can use either.

Alex
11 Feb 2011 1:38 PM

It's quite funny. I had "invented" quite the same algorithm a couple of years back when we needed to detect blobs of emission in astronomical images.