RECURSIVE METHOD TO EXTRACT RECTANGULAR OBJECTS FROM SCANS

Cormac Herley

Microsoft Research

ABSTRACT

In scanning photographs, receipts or other small objects users will often scan many at a time. It would be convenient to automatically detect that the scanned image consists of many small objects rather than a single large one, and segment appropriately. We present a simple, efficient and robust way of doing this.

1. INTRODUCTION

The problem we address is simply stated: given an image determine whether it contains rectangular objects, and if so determine their vertices. The rectangles can be of any sizes, at any positions and orientations and there can be any number of them in the image. We merely assume that they do not overlap with each other. The most obvious application is to scanner images, so that multiple photographs, receipts, or business cards might be segmented and stored automatically. We are primarily interested in objects on a constant color background. For example, most scanners have a constant black, white or grey color, but we also accommodate any other background that is predominately of one color. Let us be clear that we are not solving or addressing the compound document analysis problem: we do not seek to separate images from text on scans of magazine pages for example.

Given that we seek rectangles the obvious approaches might be to seek lines and/or corners in the image and match them up. While this might be feasible in certain cases it can become very complicated as the number of objects increases. Firstly, detecting either lines or corners at arbitrary positions and orientations can be computationally expensive. The Hough transform [1], a standard method for determining the existence of lines in an image, requires many operations and is error prone. Secondly, certain images will have one or more edges that are hard to distinguish from the background (e.g. photographs of snow scenes scanned on a white background). Thirdly, there is often sharing of lines among objects: two or more objects at the same orientation can often be confused as one.

In the next section we examine the question of estimating the background color. In Section 3 we show that when an image contains only a single rectangular object the vertices are easily determined. In Section 4 we give an example of how a recursive divide and conquer technique can simplify the multiple object case to many single object cases, and in Section 5 give the core of the algorithm. In Section 6 we examine implementation details, and how our algorithm fares when faced with real data that deviates from our idealized assumptions.

2. DETERMINING THE BACKGROUND COLOR

A key assumption is that the background is a single color and that most of the interior pixels of the rectangular objects differ from that color. We will not assume that this background is known, but rather estimate it from the data. Most of the algorithms to estimate the background color involve taking a histogram of the image or parts of the image and seeking well defined peaks. The background color, being constant, will often form a larger peak than any of the other colors.

In many cases this simple scheme is adequate; however it can fail when:

- The rectangular objects themselves contain large amounts of constant color (e.g. saturated white in snow scenes or black in underexposed images)
- The rectangular objects cover so much of the image that the number of background pixels is insufficient to form a discernible peak.

Since correctly determining the background color is of paramount importance we actually use a more sophisticated algorithm than the simple histogram approach. In the interest of space we omit the details, and will from here on assume that the background color has been accurately determined. Interested readers are referred to [2] for details.

3. SINGLE OBJECT CASE

We now demonstrate that if the image consists of a single rectangular object the problem is easily solved. The situation is as shown in Figure 1. We assume that the background color b has been estimated, and that at least a majority of the pixels in the interior of the rectangle differ from b by more than a threshold amount. Call any pixel for which $|Im(i,j) - b| >$ threshold a data pixel, and all others background pixels. Suppose we calculate the

number of data pixels in the j-th row P(j), and the i-th column Q(i). These functions have been plotted in Figure 1. Observe that for the first and last few rows we have P(j) = 0, since there are no data pixels at the top and bottom of the image. P(j) becomes non-zero at rows a and d (the rows that contain the corners of the rectangular object), and ramps from there to it's maximum width between rows b and c. Similarly Q(i) has a trapezoidal shape as also shown in Figure 1.
Elementary geometry gives that the top part of the graph of P(j) is equal to x Cos(theta) and of Q(i) is y cos(theta), where x and y are the dimensions of the rectangle, and theta the angle at which it is oriented. The corners of the rectangle are the four points (g,a), (h, c), (f,d) and (e,b) which correspond to the inflexion points of the trapezoids P(j) and Q(i).

There is actually a second possible rectangular object that would produce the same P(j) and Q(i) functions, as shown in Figure 2. This rectangle with coordinates (h,b), (g,d), (e,c) and (f,a) is at the same position as the first, but oriented at angle –theta, and is the only other possible rectangle that would generate the observed trapezoids.

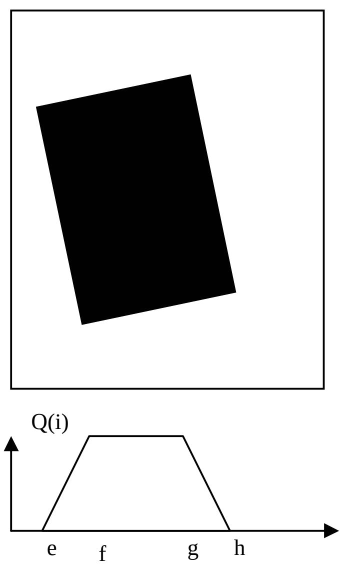

Figure 1: A single rectangular object. We graph P(j) and Q(i), the number of data pixels per row and per column respectively. Observe that the knee points of the trapezoids give the vertices of the rectangular object in the image.

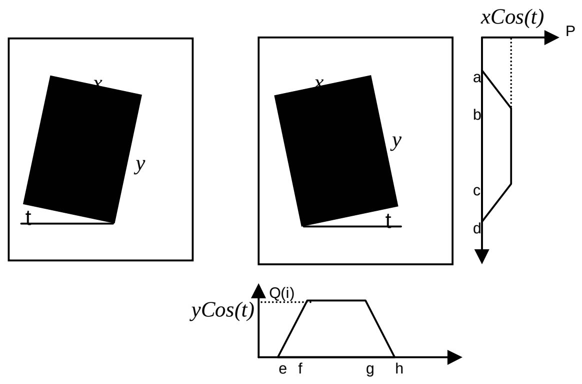

Figure 2: Two rectangular objects at different orientations give the same P(j) and Q(i) functions. To distinguish between these two possibilities one can explicitly check which rectangular object is present.

Clearly, in this simple case, if we knew that our image consisted of only a single rectangle, and there were no risk of confusing data pixels and background pixels, then we could calculate P(j) and Q(i), estimate the inflexion points (a,b,c,d) and (e,f,g,h) and then explicitly check which of the rectangles is present. We defer until later discussion of departures from this ideal case; for example where not all interior pixels satisfy |Im(i,j)-b| > threshold, or the knee points are not well defined, or the background is not of uniform color.

The kernel of our algorithm is a method that, given the functions P(j) and Q(i) calculated over that image or sub-image, determines the inflexion points, and from this determines whether either of the two possible rectangles is a plausible fit to the data. If either of them is then it returns the vertices of the rectangle. For notational convenience, let's call a routine to accomplish all of this singleObject(Im, P, Q), which takes as arguments a portion of an image, and the P(j) and Q(i) functions as defined above.

4. MULTIPLE RECTANGLES: EXAMPLE

When the image consists of multiple rectangles the situation becomes a great deal more complicated. The quantities P(j) and Q(i) will now consist of the sums of the trapezoids generated by each of the individual rectangles. See for example Figure 3, where three rectangles are present. In the ideal case it might be possible to estimate the parameters if one knew, or guessed, that three trapezoids were present. But such an approach is unlikely to be robust when faced with real data, and will become very complicated as the number of rectangles (and hence trapezoids) increases.

Observe from Figure 3 however, that P(j) contains a gap at row j_0, that is there is a location where P(j) is zero. This indicates that there is no image data at this location, and the problem can be decoupled to examine the portions of the image above and below row j_0 separately. This is an important simplification as it allows us to split the problem in two.

Having discovered this gap, we can decouple the parts of the problem above and below j_0. This is shown in Figure 4. Let's calculate the quantities P(j) and Q(i) over the two parts of the image (the rows above j_0 and the rows below). We see from Figure 4 (a) that the part above consists of a single rectangle so P(j) and Q(i) end up being simple trapezoids. Thus this subproblem is solved by routine singleObject(). The part below j_0 consists of two rectangles and P(j) is the sum of two trapezoids. The handling of this sub-image is shown in Figure 4 (b). Observe that now there is a gap in Q(i) at location i_1 indicating again that this sub-image can be broken into even simpler sub-images by taking those columns to the left of i_1 and those to the right. Those sub-sub-images each contain a single rectangle and their P(j) and Q(i) functions are simple trapezoids, allowing them to be solved using routine singleObject().

The example shown in Figures 3 and 4 illustrates that a gap in the P(j) or Q(i) function of the image will allow us to break the problem into sub-problems, and even these sub-problems can often be similarly decomposed. In this example the simplification carried us all the way to sub-images that each contained only a single rectangle and could be solved using function singleObject().

It is not the case that the simplification always leads to sub-images that each contain a single rectangle. However, every simplification makes further simplifications more likely. In the majority of cases where a user places objects on a scanner, the algorithm simplifies all the way to sub-images containing single objects. Figure 5 shows examples of several images that can be simplified and one that cannot. Further details are in [2].

5. RECURSIVE DIVIDE AND CONQUER

Our approach will be to calculate P(j) and Q(i) for a given image. If gaps are found, we simplify and re-apply to the sub-images until no further simplifications are possible. At the lowest level we have sub-images for which the P(j) and Q(i) functions contain no gaps. We then use routine singleObject() and if a single rectangle is found add it to the global list. Otherwise we decide that no rectangle has been found in the given sub-image. In either case we proceed with the other sub-images until no more sub-images remain. Because we apply much the same processing to an image and its sub-images the overall algorithm is efficiently implemented recursively.

```
function procMult(Im);
[P, Q] = getProjections(Im);
[gP, gQ] = getGaps(P, Q);

if ( #(gaps in P) + #(gaps in Q) == 0)
    singleObject(Im, P, Q);
else
    for m = 1 to #(gaps in Q)
        for n = 1 to #(gaps in P)
            procMult(Im(gQ(m):gQ(m+1), gP(n):gP(n+1))
        end
    end
end
```

Where the called functions are as follows: getProjections(Im) is a routine to calculate P(j), Q(i) over sub-image Im, getGaps(P,Q) determines position of any gaps in P(j), Q(i), and singleObject(Im, P, Q) examines P(j) and Q(i) for a single rectangular object and adds to the global list if found.

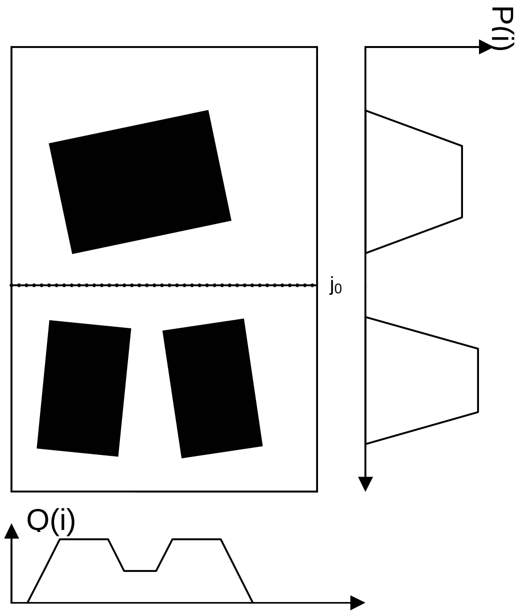

Figure 3: Example of three rectangular objects. The P(j) and Q(i) functions are now the sums of trapezoids. Observe that P(j) has a gap at row j_0, indicating that the problem can be simplified into sub-problems above and below this row.

6. IMPLEMENTATION ISSUES

We've dealt primarily with the case where there is no confusion between background pixels and data pixels. Of course this idealization seldom holds in practice. For example if the background color is saturated white then any pixels interior to the rectangles will be confused as background, resulting in P(j) and Q(i) functions that deviate from perfect trapezoids, even when only a single object is present. The main role of P(j) and Q(i) is to show where gaps occur; only approximate location of the knee points is needed in singleObject() and once we know a sub-image contains at most a single object it is possible to estimate its vertices with care. Tested on hundreds of groups of scanned photos, receipts and business cards the algorithm behaves very robustly. Figure 6 shows an example.

Being recursive, the algorithm is inherently efficient, with little duplication of effort. A detailed analysis of complexity is in [2].

[1] A. Jain, "Fundamentals of Image Processing," Prentice-Hall.
[2] C. Herley, "Recursive Extraction of Objects from Images," in preparation.

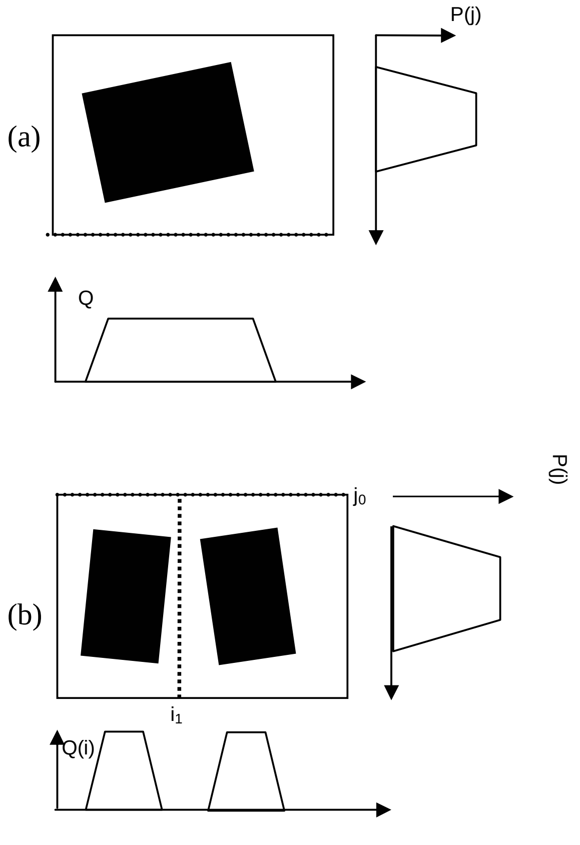

Figure 4: decoupling the example of Figure 3 by splitting above and below row j_0. We recalculate P(j) and Q(i) for the new sub-images. (a) We find the sub-image above j_0 is now the single object case (b) The sub-image below j_0 now has a new split in Q(j) at column i_1 (allowing further simplification).

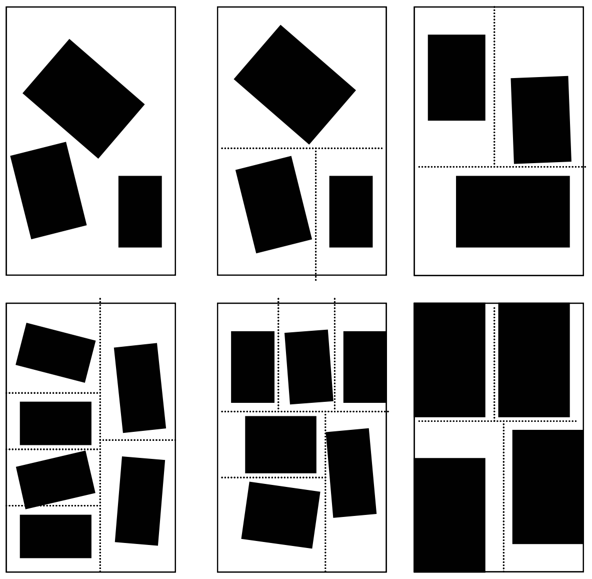

Figure 5. Examples of various configurations of rectangular objects. All but the top left example are simplified to sub-images that contain a single object. The splits generated by our algorithm are shown as dotted lines.

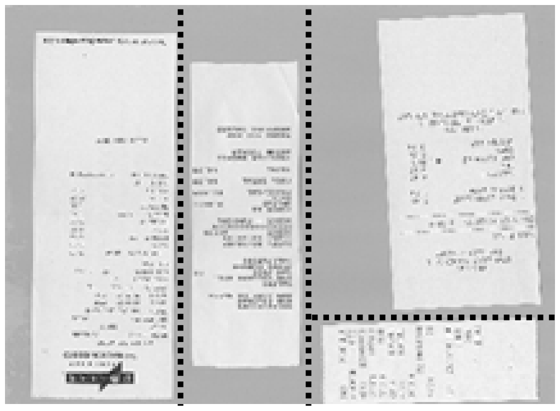

Figure 6. Actual segmentation of a scanned image. The image consists of four rectangular objects; the dotted black lines indicate the splits generated by our algorithm. All vertices were correctly found.