

# Detecting Repeated Shapes In Images

Charles Dietrich

CS 478 Final Project, May 20, 2004

## 1 Abstract

An important visual capability is the ability to quickly detect the presence of multiple similar shapes in an image. We seek a fast method to either determine that no repeated objects are in a image, or to determine that there are repeated objects and identify them. We term this the perceptual clustering property. We segment an image and present a heuristic for the similarity between image segments, and then cluster segments. For segmentation we use the fast MST-based image segmentation presented by Felzenszwalb & Huttenlocher [1]. We define a metric space for segments based on simple per-segment measures including color, size, rough shape etc. Over this space we cluster using a modified hierarchical agglomerative clustering (HAC) algorithm based on shortest distance between clusters. The running time is dominated by the segmentation algorithm, which is itself quite fast; run times are on the order of a few seconds. We find that our clustering does indeed capture the perceptual clustering property for a wide variety of images. Finally we present future directions for improvement.

## 2 Introduction

An important visual capability is the ability to recognize a plurality of similar objects in an image. We call this the perceptual clustering property in this paper. Examples of naturally occurring images with repeated objects include dandelions in a field, windows on a building, cars in a parking lot, and people in line. In many cases, for example windows on a building, the affine transformation from one instance to the next provides important clues as to geometry [2]. We note that intuitively it seems that we as humans first identify plausible repetitions and then apply more advanced discrimination; for example in a quick glance from above at a field containing both yellow dandelions and yellow tennis balls, one might (erroneously) cluster both classes of objects together; however with closer examination (or better vision!) a human would easily differentiate the two classes. I mention this because it seems a logical progression to go from simple object models to more complicated ones, in our pursuit of perceptual clustering. The algorithm presented here is a fairly simple model; though it endeavors to “see” in some real sense, it cannot hope to perform the sort of clustering that

enables a person to identify a plurality of people standing in line, and see each person as a distinct object.

Now that we have defined the problem, and demonstrated its importance, we turn to the solution. It is well established that given a model  $M$  of an object class  $X$ , we can find an instance  $x$  of  $X$  in a scene (examples: Hausdorff distance[7], Eigenfaces[8], Blobworld[4]). Given an object class  $X$  for which we wish to find instances  $x$  in an image, creating a model  $M$  for the object is itself a tricky business; it is commonly accepted, at least in academic circles, that a program that matches  $X$  should be accompanied by some semi-automated way to train  $M$  (e.g. [8][4]). However the problem as we present it here has a “chicken and egg” component to it: we don’t know what object class  $X$  we are looking for. Since we cannot have models  $M_i$  for every class  $X_i$  of objects (obviously there are way too many classes of things out there), we must somehow have a space of possible models. At a minimum we should be able to say that 1) there are objects  $x$  and  $y$  in the scene and 2) objects  $x$  and  $y$  do or do not match the same model  $M$ , i.e. they are or are not both of class  $X$ . So as a practical matter we must have a) a model for what an object is, and b) some way to compare these objects.

Leung & Malik [2] is an example of prior work in this area, though their goals were less general.

### 3 Implementation

The obvious way to establish what objects are in a scene is by image segmentation, i.e. a partition of the image. In passing we note that we are looking for objects; and in two dimensions objects can be occluded, such that a human naturally “fills in” the occlusions, so a partitioning may not be appropriate. However we do not consider this further here. Segmentation is an “application” of image labeling, where the label of a segment is the one that is most appropriate to that segment.

The canonical way to define a good labeling is to define a data cost  $D((l, p))$  of assigning label  $l$  to pixel  $p$ , and an adjacency cost  $V((l_1, p_1), (l_2, p_2))$  for pixels  $p_1$  and  $p_2$  adjacent and with labels  $l_1$  and  $l_2$  respectively. Given a (compact) representation of  $V$  and  $D$ , the problem is to find a labeling  $L = ((l_1, p_1), \dots, (l_n, p_n))$  that minimizes

$$\min_L \sum_{p_i} D(l_i, p_i) + \sum_{p_j \text{ adjacent to } p_i} V((l_i, p_i), (l_j, p_j)) \quad (1)$$

For most reasonable  $V$  and  $D$  and with more than 2 possible labels, this is an NP-hard problem. This is a little off topic, I mention this mainly to get across the idea that even with a reasonable definition segmentation is computationally hard. Secondly, this gives us a reasonably idealized baseline to compare to what we will use. There are many approaches to segmentation, for a bibliography see [1], for a much different example see [3] and for a somewhat jaded perspective on the hope of finding an ideal segmentation algorithm [4].

The segmentation algorithm we use is that of Felzenszwalb & Huttenlocher[1]. It is

related to Kruskal’s MST algorithm. For those of us who get the various MST variant names confused, Kruskal’s orders the edges in increasing weight and goes through the edge list in this order, selecting the edge if it connects disconnected segments of the graph. Thus in contrast to Prim’s algorithm at any one point there may be several non-singleton trees present. The Felzenszwalb & Huttenlocher algorithm has an interesting twist, however, that terminates the algorithm before we get one tree, so that we end up with a forest of tree components  $C$ . In their algorithm, each pixel has an undirected edge with each of its eight neighbors (the cardinal directions plus the diagonals). We define an internal difference  $Int(C)$  of a component  $C$  as the maximum weight edge in  $C$ . We also have an added resistance function or threshold function for  $C$   $\tau(C)$ . When we encounter an edge  $e$  that connects components  $C_1$  and  $C_2$ , we add  $e$  if the weight of  $e$  is less than  $Int(C_1) + \tau(C_1)$  and less than  $Int(C_2) + \tau(C_2)$ , i.e. it is not much more than the internal difference of each of the components. The function  $\tau$  can be arbitrary but in their implementation is  $k/|C|$  for a constant  $k$ , which establishes a preference for larger components.

This algorithm satisfies certain properties and is in a strong sense “deterministic” or “non-local search”. As the authors note it enables noisy regions and gradient regions to be identified properly, and in this sense captures a notion of “perceptual grouping”. In contrast to our canonical segmentation cost (1), which implies a boundary cost, here we have no real notion of a boundary cost, so that in particular a path of small edge costs can join two dissimilar regions. Practical examples of this limitation are in [5].

The algorithm has the attractive property that its running time is nearly linear and it is fast in practice (for our system, about a second). As a practical matter, segmentation was done on each color channel and the final segmentation obtained by intersecting the segmentations from the three color channels (as in [1]).

We used the code from [1] (available on the web) as the basis for our project.

Once a segmentation was found, we set about comparing the objects. We found that we got the best results with few ( $s \simeq 50$ ) segments, so that direct pairwise comparison of segments was possible (i.e. we had  $s^2$  edges). However, we defined a segment as a point in an  $n$ -dimensional space  $P$ , so in theory we could use approximate nearest neighbor algorithms to define  $O(s)$  edges. We used the image moments in [6] to calculate a rectangle that has the same image moments as the segment; the rectangle is defined as the centroid of the segment, an attitude  $\theta$ , and the lengths of the two edges  $l_1$  and  $l_2$ .

Each segment was assigned a point in  $P$  according to the following attributes (dimensions of  $P$ ):

- position of centroid  $c$  (x,y)
- size of component
- average color (R,G,B)
- maximum internal color variation (R,G,B), defined as for each color channel  $a$ , the difference between the maximum value and the minimum value in that component.

This roughly corresponds to  $Int(C)$  defined above, except that our segments are the intersection of components in each of the three color channels (so that our  $Int(C_a)$  is at most the original  $Int(C_a)$ ).

- eccentricity, defined as  $\frac{\max(l_1, l_2)}{\min(l_1, l_2)}$
- $\theta$
- the mean distance squared for the segment  $C$ , defined as  $\frac{1}{size} \sum_{p \in C} \|p - c\|^2$  (thanks to Pedro Felzenszwalb for suggesting this measure).

To weight each dimension, a dilation matrix  $D$  was applied to  $P$ . (This can be set via the command line.)

To segment the image, we used a variant of HAC. Conveniently enough, we had an MST algorithm from [1] (i.e. by setting  $k = \infty$ ). So we used the variant of HAC that uses shortest-distance to merge two components. We modified the existing code to implement a maximum edge cost *cutoff* (settable via the command line). We kept the parameter  $k$  that defines  $\tau(C) = k/|C|$ , since this is actually appealing for our problem. For example, if we had a tight cluster in the feature space we might not want to “dirty” it with not so closely related objects. For example, consider one tennis ball in a field of dandelions. Obviously the  $k$  used for clustering and the  $k$  used for segmentation may be different.

## 4 Code

As noted above, this was built upon the code provided for [1]. Our code runs on linux. Usage is as follows:

Usage:

```
segment [ -<option> ... ] input
```

Input:

```
file      Input file name.
```

Options:

Parameters:

Segmentation Options:

```
-s X      sigma:                               (Default = 2.000000)
```

```
          sigma of gaussian filter used to smooth the input image  
          before processing.
```

```
-k X      Threshold k:                         (Default = 500.000000)
```

Constant k for the threshold function.

-m X Minimum Size: (Default = 30)  
 If different than zero a post-processing step will eliminate components smaller than this size and grow the remaining ones.

Cluster Options:

-Ck X Threshold Ck: (Default = 1000.000000)  
 Constant k for the threshold function.

-Cc X Cutoff Cc: (Default = 300.000000)  
 Maximum edge weight (cutoff).

-Cp X Position Cp: (Default = 1.000000)  
 weighting for proximity.

-Cs X Size Cs: (Default = 15.000000)  
 weighting for size.

-Ca X Average Color Ca: (Default = 15.000000)  
 weighting for average color.

-Ci X Internal difference Ci: (Default = 5.000000)  
 weighting for internal difference (in color).

-Ce X Eccentricity Ce: (Default = 1.000000)  
 weighting for eccentricity.

-Ct X Dominant direction Ct: (Default = 0.500000)  
 weighting for dominant direction theta.

-Cm X Mean distance squared Cm: (Default = 1.000000)  
 weighting for mean distance squared (size invariant).

-o X Output: (Default = seg.pnm)  
 Name of output files.

The program can handle RGB images in p\*m format of any size, though I used 240x320 images.

The following files are produced (in RGB p\*n format):

- original: the original image.
- seg: segmentation files, false colored. This is the output of the original program from [1].
- cluster: each cluster of segments has a false color, and the background (i.e. belonging to no cluster) is greyscale. One limitation of this is that it is not possible to differentiate adjacent segments from one big segment, without referring back to seg.
- pop: segments belonging to a cluster are in color, the rest of the image is in greyscale.

## 5 Experimental Method

I took a bunch of images with a Sony Mavica 1.6 Megapixel camera (“Russ”, borrowed from the Accel computing lab). All images were scaled to 240x320 and converted to the p\*m format. Images were taken on two different days. I attempted to take pairs of images that were of the same thing, but under different lighting conditions (for example `cobblestones1` and `cobblestones2`), in order to gauge the robustness of the algorithm.

There was some fiddling with the parameters in order to get good segmentations and good clusterings. I intended to train on a small set of images, but in practice I found a good setting for `dandelions` that worked on the remainder of the training samples noted below, so I went with that.

Test Images:

- `dandelions`
- `bollards1`
- `bike-racks-and-windows`
- `flowers`
- `upson-floor-dots1`

The remainder of the images are training images, in that I did not observe their output before the program was completed and the parameters set. As a sidenote, 3 additional images were used but caused the program to crash, so they were deleted from the test set.

The images and the output are in the appendix, and also online at <http://www.charlesdietrich.com/countdracula/>

I did not have time for a formal protocol for judging the efficacy of the program.

## 6 Discussion

I consider the output good overall. However a lot of edges were detected as matching, possibly due to stringlike clusters along the eccentricity axis (recall that stringlike clusters are a weakness of the segmentation algorithm because of its reliance on  $k$ ). This brings up the point that it would probably be better to use mean (centroid) distance in the HAC algorithm, though this creates an NP-hard problem if used with  $k$  (without  $k?$ ).

Sometimes there were apparently artificial cluster boundaries, for example with the upson floor tiles.

Finally, [1] describes a non-local segmentation that clusters pixels in  $(r,g,b,x,y)$  space and uses approximate nearest neighbor to establish edges between pixels. It would be interesting to see if a properly tuned version of this algorithm would produce results similar to those shown here.

## 7 Future Work

I could establish ground truth by segmenting images and hand-selecting a good segmentation; from this segmentation I could hand-select clusters. From this a protocol to evaluate the quality of the solution could be devised. The ground truth could also be used to automatically find a good discriminant matrix  $D$ , using the criteria that clusters should be as dense as possible and have as much margin as possible from non-cluster segments.

Given that a small number of segments seems to work best, it would be good to have a pairwise comparator between segments, for example silhouette matching. Obviously a non-linear space could also be used.

The Blobworld paper [4] describes a *per-pixel* texture and texture orientation metric. It would be interesting to apply this both to the initial problem in [1] and to our problem here. (Obviously a per-segment texture metric can be defined.)

Additionally, [4] uses an Expectation Maximization algorithm for segmentation. As I noted above, there are many segmentation algorithms; it would be interesting to try this problem with a different segmentation.

An idea that would be easy to implement is to run the segmentation several times with different parameters and select the segmentation that we “like best”. This was the appeal of choosing a fast segmentation algorithm. This could be as easy as selecting a segmentation with a certain number of segments. In practice this didn’t seem necessary.

## 8 Bibliography

- [1] P. F. Felzenszwalb and D. P. Huttenlocher, “Efficiently computing a good segmentation”, DARPA Image Understanding Workshop, 1998.  
<http://citeseer.ist.psu.edu/felzenszwalb98efficiently.html>
- [2] Leung, T.K., and Malik, J., “Detecting, localizing and grouping repeated scene elements from an image,” (1996) Fourth European Conference on Computer Vision, Cambridge, UK, Vol 1, pp. 546-555.  
<http://citeseer.ist.psu.edu/leung96detecting.html>
- [3] Y. Boykov, O. Veksler, R. Zabih, “Fast Approximate Energy Minimization via Graph Cuts,” Intl. Conf. on Computer Vision, 1999.  
<http://citeseer.ist.psu.edu/article/boykov99fast.html>
- [4] C. Carson, S. Belongie, H. Greenspan, and J. Malik. Blobworld: Image segmentation using expectation-maximization and its application to image querying. 1999. (in review).  
<http://citeseer.ist.psu.edu/article/carson99blobworld.html>
- [5] D. Huttenlocher. CS 664 Slides #11. Lecture notes for CS 664, Cornell University, Fall 2003.  
<http://www.cs.cornell.edu/courses/cs664/2003fa/handouts/664-l11-segmentation-03.pdf>
- [6] Freeman, W., Anderson, D., Beardsley, P., et al. Computer vision for interactive computer graphics. IEEE Computer Graphics and Applications, Vol. 18, Num 3, pages 42-53, May-June 1998.

<http://citeseer.ist.psu.edu/freeman98computer.html>

[7] P. F. Felzenszwalb and D. P. Huttenlocher. Pictorial structures for object recognition. Submitted to IJCV, 2003.

<http://citeseer.ist.psu.edu/felzenszwalb03pictorial.html>

[8] M. A. Turk and A. P. Pentland. Face recognition using eigenfaces. Proc. CVPR'91, pages 586–591, 1991.