Efficient Graph-Based Image Segmentation

CS 534 Project, Fall 2015

Dylan Homuth and Coda Phillips

Abstract: This paper details our implementation of a graph based segmentation algorithm created by Felzenszwalb and Huttenlocher. The algorithm represents an image as a graph and defines a predicate to measure evidence of a boundary between two regions. This is highly efficient running in $O(n \log(n))$ however it comes with a trade off in strict accuracy. We tested our implementation using the Berkeley dataset which established a standardized scoring method to better compare segmentation algorithms to a ground truth segmentation created by humans.

Website: http://pages.cs.wisc.edu/~coda/cs534_project/
Introduction
Theory
Method
  Pre-processing
  Graph Construction
  Segmentation
  Human Input
Results
  Performance
  Precision/Recall
Soft Edge Comparison Results
  Good Results
  Bad Results
Conclusion
References
Introduction

There are many devices and image processing algorithms that require the identification of unique objects and regions. It appears in applications for medical devices, self driving cars, surveillance, and much more. Image segmentation algorithms have been developed specifically for many of these cases, but there is not one general solution to image segmentation that can compete with humans. Part of the reason the problem is still open is because it is not well-posed problem. There is no unique segmentation for any image, even humans don't always choose the same segmentation for a given image. The paper we followed for our project was attempting to create a segmentation algorithm fast and robust enough to use as a general tool in image processing.

Theory

The idea behind Felzenszwalb and Huttenlocher’s paper was to use a highly efficient graph based approach. Each pixel in an image was represented as a node in a graph with edges connecting pixels to their neighbors. Each node may have many edges, possibly to nodes not exactly adjacent, but for our project we used a 4-connected graph. Each node is connected to the pixel above, below, left, and right of it. The edge weights are a measure of pixel dissimilarity, the more different two adjacent pixels are, the greater the edge weight. This dissimilarity can be a number of things, however the most general is simply the intensity values on each of the color channels which are then combined.

The graph needs to then be divided into segments. Each node starts as it’s own segment and is then joined with others to form visually distinct parts. For every edge in the graph a decision must be made whether to join the segments on either end of the edge. To make the decision the paper defines a predicate, which takes two segments or components as input and outputs whether there exists evidence of a boundary between them. It does this by comparing the smallest edge connecting the two segments and the largest edge within each segment. In the case of merging, this means that if the most similar pair of pixels on the border of the two segments is more similar than the least similar pair of pixels already in a segment, you should merge the two segments. Conversely, if a pair is less similar than the least similar internal pair, you know the two regions form a boundary.

Method

Pre-processing

We used a median filter with a 3x3 kernel size to pre-processes every image. The purpose of pre-processing is to remove digital compression artifacts and noise from the image which will harm the results of the segmentation. We used JPEG images for input, so this was important in reducing the inherent noise created in compression.

Graph Construction

To construct the graph we iterated through all pixels adding between zero and two edges per pixel. The first edge would start at the current pixel and go to the next pixel along the row on the right. If the current pixel is on the rightmost column then this edge doesn’t exist. The second edge would start at the current pixel and go to
the next pixel in the column, below the current one. If the current pixel is on the bottom edge then this edge doesn’t exist. For each edge, the corresponding edge weight is calculated by taking the Euclidean distance between the endpoints in RGB-space. Edges were stored in a python dictionary, implemented internally as a hash map with $O(1)$ lookups. A tuple of start-end coordinates served as the key with the value being the edge weight. There was an attempt to use a high-performance ordered dictionary with sorted insertions, but little benefit was gained over a $O(n \cdot \log(n))$ sort at the end.

Segmentation

The segmentation algorithm begins by sorting edges by edge weight. An empty defaultdict mapping for holding information pertaining to components is initialized. The disjoint-set data structure is initialized to be empty.

Next the edge with the smallest weight is retrieved. The labels for the nodes at either end of the edge are found in the disjoint-set data structure using a find method. This method should complete in worst-case $O(\log n)$ and return a label unique to the component containing each node. It is possible that both nodes are already part of the same component and the labels match. In this case the edge may be skipped. If the labels differ then the pair of components are either unified or left disjoint depending on the output from the predicate.

The predicate requires knowledge of the maximum-valued edge of the minimum spanning tree for each component. Calculating minimum spanning trees for each iteration of the algorithm would be slow. Instead one can observe that the maximum-valued edge for a component’s minimum spanning tree is the last edge added to that component. By storing this value in the defaultdict mapping for the next iteration you can dramatically increase performance. Size of the component is another required input to the predicate, this may be stored and incremented in the same mapping as the maximum-valued edge. The predicate also requires the value of the minimum-valued edge connecting the two components. By inspection this is always the current edge under examination. Computing the predicate using the information saved in the defaultdict can be performed in $O(1)$ time.

If the predicate returns false, then there is evidence for a boundary between the components and the algorithm may repeat with another edge while one exists. In the case that the predicate returns true, no evidence of a boundary has been detected and the components will be merged. In practice the labels of the smaller subset of nodes will be reassigned to the labels of the larger subset. Using the union method of the disjoint-set data structure the worst-case running time is $O(\log(n))$. The cached size and maximum value for each component will be deleted and replaced with a single entry for the new larger component. Once this is done the algorithm will repeat until there are no larger edges.

Human Input

The algorithm is not completely automated. A value, tao, modifies the predicate’s value for internal difference to measure some characteristic of the segment. This could be a measure of size, shape, location, or anything you can use to define what you are looking for in a “good” segment. We used size as our factor which scales with the size of the regions, getting smaller with larger regions, making it more difficult to combine segments as they get larger. The threshold can vary wildly based on the image, depending mostly on size of the image and size of the objects in the image. Currently, human input to this value is the only way to adjust to an optimal segmentation.
Results

Performance

Through use of performant data structures and clever optimization, *Efficient Graph-based Segmentation* (EGBS) achieves performance that is \(O(n \cdot \log(n))\) in number of pixels. The main limitations are the speed that the edge weights may be sorted, the \(n\) \(O(\log(n))\) lookups for node components and the \(n\) \(O(\log(n))\) unions. There is the possibility of parallelizing and exploiting multiprocessor systems, this could be accomplished using Helper Threads (Anastasios Katsigiannis et al.) but any improvement would probably only be seen in very large images.

Although EGBS performs in almost linear time on the number of pixels, the number of pixels increases with the square of image size. An image that is only 40% bigger on each side almost doubles the running time.

Performance is also dependent on image content, it is possible for images of the same size to segment in different amounts of time. The running time decreases as the number of segments in the final segmentation increases, with size being constant. This is because fewer and larger segments correlate with additional unions; each extra union requiring extra time. The size of individual segments can be influenced by a user-supplied “k” parameter. This offers some control, but the number of segments can still vary. *Figure 2* shows a histogram of run-times for the Berkeley Segmentation Dataset [BSDB]. All images are the same size and all have identical \(k\) values, yet the values can vary up to 10% from the mean.
Precision/Recall

EGBS was benchmarked on the Berkeley BSDS300 segmentation dataset using the Berkeley Semantic Boundaries Dataset and Benchmark (SBD). The Berkeley data set was created to consistently judge and accurately compare different segmentation algorithms. It quantifies human segmentation for a large set of images and holds this as the ground truth. A soft boundary map for a testing algorithm is compared to the true segmentation, measuring two criteria: precision and recall. Precision is a measure of how much noise or errors are in the output of the detector. Recall is a measure of how much of the ground truth is detected. The curve’s below shows the inherent trade-off between these two quantities, misses and false positives, as the detector threshold changes. The left figure demonstrates various established algorithms while the one on the right shows the overall results of our implementation of EGBS. Our implementation performed only slightly better than random segmentation (see table 1) on average. However, there were cases where the algorithm actually performed better than humans.

Figure 3 Evaluation of segmentation algorithms of the BSDS300 Benchmark. Image credit to P. Arbelaez et. al.

Figure 4 Evaluation of EGB segmentation.
<table>
<thead>
<tr>
<th>Score</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.79</td>
<td>Humans</td>
</tr>
<tr>
<td>0.71</td>
<td>Ren et al. NIPS2012 (color)</td>
</tr>
<tr>
<td>0.71</td>
<td>gPb-ucm (color)</td>
</tr>
<tr>
<td>0.70</td>
<td>Global Probability of Boundary (color)</td>
</tr>
<tr>
<td>0.68</td>
<td>xren (color)</td>
</tr>
<tr>
<td>0.67</td>
<td>Arbelaez POCV2006</td>
</tr>
<tr>
<td>0.66</td>
<td>Boosted Edge Learning (color)</td>
</tr>
<tr>
<td>0.65</td>
<td>min-cover</td>
</tr>
<tr>
<td>0.65</td>
<td>Brightness / Color / Texture Gradients</td>
</tr>
<tr>
<td>0.63</td>
<td>Brightness / Texture Gradients</td>
</tr>
<tr>
<td>0.57</td>
<td>Color Gradient</td>
</tr>
<tr>
<td>0.55</td>
<td>Efficient Graph-Based Segmentation</td>
</tr>
<tr>
<td>0.43</td>
<td>Random</td>
</tr>
</tbody>
</table>

*Table 1 [BSDB] Comparison of Algorithms*
Soft Boundary Map Comparison Results

Below are some of the results from running our implementation on the Berkley training set. The first image is the original testing image, the second is the soft boundary mapping of the segments produced by our algorithm, and the third image shows the ground truth it is comparing to. The graphs show the precision and recall amounts for our algorithm with varying human input values and the results of several human segmentations. You can see that in some cases our implementation outperformed humans in some test cases but for others was significantly worse.

Good Results
Bad Results
Conclusion

The efficient graph based segmentation is very fast, running in almost linear time, however there is a trade off. We lose a lot of accuracy when compared to other established segmentation algorithms. The authors' intentions were to create a segmentation tool which could be used efficiently to assist with other functions. Depending on the function the efficiency trade off might be worth it, but in a general case this is not the ideal solution.
References


Contour Detection and Hierarchical Image Segmentation
P. Arbelaez, M. Maire, C. Fowlkes and J. Malik.

Code we used:

We used a union-find data structure from the python_algorithms package.
https://pypi.python.org/pypi/python_algorithms

Code we wrote:

We wrote approximately 50 lines of Python code to accomplish the basic task (posted on website) and a couple dozen additional lines of code written for benchmarking, and testing.

Contributions:

Coda Phillips: Python programming, benchmarking, plotting, and testing
Dylan Homuth: Documentation, planning