A Comparison of Image Segmentation Algorithms

Jessie Baskauf, Gabriel Brookman, Toni Eidmann, Miriam Gorra, Henry Pearson, Bryan Richter

Carleton College

Abstract. Image segmentation is the task of labeling the pixels of an image into groups that are representative of some sort of coherent elements of the image. It is a complex task, and many algorithms exist to accomplish it, with varying degrees of success. We explore six alternatives to solving this problem and discuss their relative strengths and weaknesses, evaluating both by qualitative comparisons to ground truth images and by two different evaluation metrics.

Keywords: image segmentation, thresholding, region growing, split and merge, k-means, watershed, minimum cut

1 Introduction

Image segmentation is a process by which image pixels are grouped into different, separate sections known as segments. Image segmentation is not the same thing as image recognition, as there is no thought given to identifying what a segment is. Images are segmented in order to be easier for a computer to work with, as now instead of looking at the collection of pixels as a whole it can now work with smaller sections of pixels to achieve tasks.

Image segmentation is commonly used for many different tasks such as segmenting medical images to find abnormalities, segmenting finger print images to make it easier to recognize fingerprints, and segmenting satellite imagery to find patterns in the land. Image segmentation is also commonly the first step of image recognition, as once an image has been segmented it can become easier to recognize what each segment is, as opposed to viewing the picture as a whole.

There are multiple different algorithms to segment an image, and each algorithm has its own benefits and drawbacks. Some require specific knowledge about the images that you will segment, others are really quick but make assumptions about the data. For our project, we implemented and tested six of these algorithms. The six algorithms we worked on were thresholding, region growing, split and merge, k-means, watershed, and min-cut. Each of these algorithms is significantly different than the others, yet is still used for the same task.

2 Algorithms

In this section, we will dive deeper into the six different image segmentation algorithms we implemented. We start by looking at the simplest algorithm and move on to more complex ones. It is important to note some terminology that we use throughout our discussions of the algorithms. To start, we assume an image is an array of n pixels. An individual pixel can be referenced by defining the coordinate (i, j), where i is a row and j is a column. Furthermore a pixel has color intensity values for red, green, and blue, each ranging from 0 to 255, which are its RGB values. Different combinations of RGB values are what produce each pixel's color and make up an image. We will also consider grayscale pixel intensities, which range from 0 to 255, where 0 is black and 255 is white.

2.1 Thresholding

Thresholding is an image segmentation technique that works by selecting one or more thresholds for some pixel value, and then labeling each pixel relative to the thresholds. For example, if an image was in grayscale and you picked a threshold of a 127, you would divide the pixels of the image into two segments based on whether they are above or below the threshold. Thresholding is a simple but effective algorithm that works extremely well on simple images, but starts to suffer against more complicated images.

Picking a threshold is the most important piece of this technique, and there are multiple effective ways of doing so. Images are often converted to grayscale for thresholding, as picking a grayscale value to threshold on is relatively simple as opposed to worrying about RGB color or other pixel characteristics. With grayscale images, the first threshold selection we applied was by using the image's average grayscale value. This was a very quick and effective technique that worked well on images with very clear and obvious separate segments, but on even slightly more complicated images would get thrown off by multiple high grayscale pixel values. We then transitioned to Otsu's method for thresholding selection [5], an algorithm that creates a histogram of counts of pixels at each possible grayscale value and tests every possible threshold value to find the one that maximizes the difference between the two sides of the threshold (above and below).

Otsu's threshold selection algorithm works by minimizing the intra-class variance, described for each possible threshold t, which is a grayscale value (and it's corresponding histogram index) by:

$$\sigma^2(t) = w_0(t)\sigma_0^2 + w_1(t)\sigma_1^2 \tag{1}$$

In this function, w_0 and w_1 are both the probability of something being below and above the threshold, respectively. This algorithm checks every possible threshold and calculates this value, then finds the value for which it's minimized.

Otsu's algorithm was a bit slower than average grayscale value, but worked incredibly well on two object images. We used Scikit-Image's implementation of

Otsu's which was both quick and effective. An issue with Otsu's algorithm arises on more complicated images however, as Otsu's assumes that a histogram to be thresholded will be bimodal and thus gives worse results on other shapes. This points to the downside of thresholding, which is that it's fundamentally limited.

We focused on finding a single threshold in grayscale, but other thresholding algorithms exist that focus on color, intensity or other pixel values. There are also ways to select multiple thresholds to extend thresholding to multi-object images. Our project did not focus as much on these extensions as the more complicated a thresholding algorithm becomes, the more it loses its advantage of speed. While we observed that thresholding was really effective on simple images, grayscale thresholding performed incredibly worse as images contained more segments and became more complicated. While there are multiple other ways to threshold images than just on grayscale, the limit that all of thresholding shares is that it cares only about a single criteria with which to threshold on. In reality, it is often a combination of things that allows images to be segmented effectively, and thresholding does not account for that.

2.2 Region Growing

Seeded Region Growing is a widely used region-based Image Segmentation algorithm first created by Adams and Bischof in 1994. The algorithm begins by selecting a seed or set of seeds either manually or by some automatically generated method. There are several different approaches for seed selection, which vary in success, however, ultimately we implemented random seed selection with a single seed as the baseline method. After seed selection, a threshold is chosen or computed by some method. This threshold is a homogeneity criterion, which is then used in deciding which pixels to add to the region. For our implementation, the chosen method was Otsu's Algorithm, similar to the one used in Thresholding. This is also the method that is most commonly found in literature on region growing.

Thereafter, the neighbors of the seed pixel are found. Again, there are different choices a programmer can make, and we decided to use the 8-connected neighboring pixels. For each pixel in the neighboring region, its RGB color values are extracted and compared to the seed values. There are multiple ways to compare pixels to each other, but the one most commonly used in the literature is Euclidean distance. These distances represent how similar or dissimilar in color the pixels are to the seed. The Euclidean distance is found for each neighboring pixel to the seed or the region. The pixel that has the smallest distance to the seed is the one that is the most similar to the seed in color. This pixel is then compared to the threshold value that was already calculated. If the distance from that pixel to the seed is less than the threshold, it is similar enough to be added to the region. Thus, after the first iteration, the region consists of a seed pixel and one of the most similar neighboring pixels. This process is repeated until all of the pixels in the image are assigned to be either in the region of interest or outside of it. As a result, region growing produces a binary image, where each pixel is assigned to one of the two resulting segments.

Since there are a wide range of ways to implement region growing, our algorithm is only one particular example of what region growing can do. Specifically, there are variations that may change the results of the segmentation. For example, we decided to select a single seed for the entire image. Other implementations use more than one seed over the image and grow outward from each of those seeds in parallel to form the segmentations. This does seem to have benefits, as it may segment some images better. This is because if a seed is in a region that has strict boundaries, it may never grow to the other side of the image even though there may still be an object found. However, if there are two seeds, it is more likely that the other object will be found if the other seed is in that region. However, from our results, we found that increasing the number of seeds greatly increases the already slow runtime of our algorithm, without actually improving the results significantly. Therefore, we stuck to a single seed.

Furthermore, random seed selection is not the ideal method for choosing the optimal seed and creating the best segmentation. However, seed selection is the most important step of this algorithm. A better seed will produce a better segmented outcome. However, this was not discovered until the end of the project. For now, our algorithm may select a good seed by chance, but it also may select a very bad one that produces a horrible segmentation. Though this method is not great, it did show us how vital seed selection is to the success of region growing. After further research, we found that there are several other methods in the literature that select seeds for this algorithm in varying ways. Many of these are more complex than we had time to implement, but would be interesting to attempt in the future.

One such example uses the idea of edge detection. On a high level, this algorithm finds the edges of an image while suppressing noise at the same time. The first step of this seed selection process begins with finding the RGB values from the input color image, which we can easily find. Next, a gradient magnitude is computed of the RGB components of the image using an algorithm called the Sobel Edge Detection Operator. The gradient magnitude for each component is added producing a new color image. Using edge detection computes the gradient of the image intensity function. In other words, it shows where the image has regions of high contrasting intensity. Finally, the threshold is used to find the seed pixels. This approach is much more advanced than random selection, and would require more time than we had available to successfully implement, though it does seem straightforward enough to be used for seed selection.

2.3 Split and Merge

The region split and merge algorithm is a two-part region-based image segmentation algorithm. The algorithm actually consists of two mutually dependent processes, split and merge. The split begins by reading the image and, using some standard of homogeneity, decides if the image is to be a single region or not. If so, it repaints the image as an average gray-scale value and returns it. If not, it splits the image into four quadrants, and runs the homogeneity check on each of those, stopping once no more splits can be made.

The merge portion of the algorithm is a depth first merging of all combinations of segments that could be considered “flat”. Essentially, if two regions read as homogeneous, using the same criteria outlined before, they are merged into one continuous region, using one average color. This is done until no merges can be made.

Ideally, split will create a minimum level of detail for a given segment, in order to define the edges of what will become the final regions. Because split will invariably over-segment in this process, merge is a requisite part of the algorithm, creating the maximum level of detail for the final regions by solidifying segments that are ultimately unimportant.

This can be completed quite quickly; the O-complexity of the algorithm is $O(n \log n)$, though it can be greater depending on the implementation of merge, which is dependent on how the data from split is stored. However, split and merge’s greatest quality is the fact that it can produce an arbitrary number of segments; this gives it an additional degree of computability power regarding complex images that have multiple distinct regions. It also is quite good at not over-segmenting on “tiny” details, such as noise from objects like water and foliage, as well as gradual gradients created by soft lighting in the photo.

The major weakness of split and merge is that the ultimate top-down design can lead to both over-segmenting and under-segmenting. Images that are quite simple, such as a single object located in a corner of the image, can be read as being homogeneous, while even a significant detail that goes over quarter lines could also end up being blended into a single region, due to its details being only considered in half of two wholes arbitrarily. These issues mostly stem from the fact that split is a dumb algorithm, simply dividing the image into recursive quadrants until it no longer can without any specific discernment besides the homogeneity test. Merge, which is only slightly more discerning, is slave to whatever was produced by split. As such, much of the accuracy of the metric rests solely on the homogeneity test, which is a lot of work to press on a single part of the algorithm. It should be noted that this implementation of split and merge as referenced in this paper is not quite complete; while merge can be accomplished within quadrants, regions belonging to two separate quadrants cannot be merged. This is mostly due to the lack of time to implement a means of checking for adjacency between regions of different sizes. As such, our version of split and merge has a tendency to over-segment quite a bit.

2.4 K-Means

K-means is a general clustering algorithm with many applications, including image segmentation. This broad utility makes it well-researched, but also less effective at image segmentation than some other, more specialized algorithms. The basic concept behind k-means is to assign every data point (in this case, pixels) to an initial cluster, then iteratively improve those cluster assignments. The algorithm begins by choosing k cluster centers, then assigns each data point to the center whose color values are nearest to them. Then the new centers of these clusters are recomputed based on the averages of all points for each cluster. The algorithm then once again assigns points to their nearest center, and continues updating centers and assigning points until convergence (when the centers stop being significantly updated in each new iteration) [2].

While kmeans is a popular algorithm, it has several notable weaknesses. The most pressing is its inability to choose the number of clusters it creates, instead relying on the parameter k. This can be useful for some applications, since the user can specify the number of segments desired, and it is both flexible and doesn't risk drastically oversegmenting. However, since we wish to be able to run kmeans automatically on a wide range of images, some means of choosing k must be used. For this purpose, our algorithm uses the elbow method, which runs k-means multiple times, with increasing values of k. It then calculates the distortion of each segmentation by finding the average distance of points from their cluster center. As the number of clusters increases, the distortion will always decrease, but there is usually a visible "elbow point" after which additional clusters offer little improvement [7]. We automatically find this elbow by finding the point with the maximum distance from the line between the first and last values of k tested. This point is a good choice for k, as it finds the number of clusters where each additional cluster has significantly decreased distortion.

Kmeans runs in O(n*k*i), where i is the number of iterations until convergence, and typically rises along with k, since there are more cluster values that must converge. This means it runs quite slowly on the number of pixels in our images, especially since the elbow method requires running kmeans multiple times on each image. Our implementation took around seven minutes per image, which meant we were unable to test varying which features we used to calculate closeness, or run expectation maximization, the probabilistic variant of kmeans.

2.5 Watershed

The watershed algorithm is a popular algorithm used for image segmentation. To understand how it works, imagine a grayscale image topographically, with its lighter values representing high points in the terrain and its darker values representing lower points in the terrain. Now, imagine tiny holes are poked in all of the minima of that terrain, and then it is lowered into a bath of water. The gullies in the terrain begin to fill with pools of water that floods through the holes. Whenever the two or more pools touch, they both stop expanding in that direction rather than combining. Each of these pools becomes a segment in a watershed segmentation.

As stated, the watershed algorithm only functions on grayscale images. Furthermore, because the watershed algorithm generates one segment for each minimum in the image, it's very prone to oversegmentation when used on natural images, which tend to have a lot of minima. On the other hand, the algorithm is extremely useful for segmenting dark images with lighter colored borders between them (or vice versa), because of how it functions.

In addition, there are a number of modifications that can be made to the algorithm that aid its vulnerability to oversegmentation. In our implementation, we made use of blurring on the input image as well as a pruning algorithm based on Wolf Pruning. Blurring prevents small amounts of noise and textural variance from being treated as additional segments by the algorithm (since these things often lead to minima appearing), so by using it, we are able to correct for the oversegmentation that might occur as a result of noise and textures being present in our input images.

Furthermore, we are also using a modification to the algorithm based on Wolf pruning. This allows us to remove segments from consideration which have minima with depths that are too low – that is, if the color distance from the minimum to the nearest saddle point (in other words, barrier between regions) is lower than a certain threshold, the region corresponding to that minimum is combined with a neighboring region. This eliminates minima without significant magnitude from consideration. While based on Wolf pruning, our algorithm has some differences from it. Namely, Wolf pruning adjusts its threshold value to eliminate regions until only a certain number of regions remain, whereas our method statically removes regions with minima below a certain, static threshold.

2.6 Minimum Cut

The minimum cut algorithm that we explored is based off of a paper from 2000 by Shi and Malik [6], who optimized an earlier attempt at using minimum cuts for image segmentation by introducing a normalization strategy to counteract the original algorithm's tendency to create small segments. The basic algorithm builds on the idea that a graph can be partitioned into two components by removing edges such that the total amount of weight on the edges removed is minimal. This algorithm is commonly used in directed graphs that have a single source node and a single sink node, and can be shown to be equivalent to the problem of finding the maximum flow through such a network. However, this version of the algorithm works on an undirected graph with no specified source and sink nodes.

First, the image is converted into an undirected graph whose vertices correspond to pixels and whose edges go between pixels that are close to each other, with edge weights based on how similar two pixels are to each other. Similarity can be measured in a variety of ways. Possible values include Euclidean distance between RGB values, physical Euclidean distance in the image, or even metrics that attempt to capture more complex patterns. In this project, we only considered RGB distance and physical distance. We can still find a minimum cut in this graph, but to calculate a normalized minimum cut precisely in this scenario, we would need to consider all possible pairs of source and sink nodes, which is computationally intractable. The normalized cut algorithm attempts to alleviate this issue by solving an approximation of this problem using eigenvalue calculations.

The idea behind the normalized cut is that it acts as a measure of how well a cut dissociates between segments and how well it associates within segments. It can be shown that these two factors are dependent on each other. Given a partition of the set of vertices V in to components A and B and a corresponding cut value $cut(A, B)$, the cut value is normalized based on the level of association (weight on all the edges) between the vertices in each of the components and all of the vertices in V (see Eq. 2). The $Ncut$ value for this partition of the vertices can be computed to measure how good that particular partition is (see Eq. 3), but finding the minimum $Ncut$ value out of all possible partitions is NP-hard.

$$asso(A, V) = \sum_{u \in A, t \in V} w(u, t) \tag{2}$$

$$Ncut(A, B) = \frac{cut(A, B)}{asso(A, V)} + \frac{cut(A, B)}{asso(B, V)} \tag{3}$$

Fortunately, this minimization problem can be solved by finding particular eigenvalues and eigenvectors for a matrix that is constructed based on the adjacency matrix of the graph. This calculation can be approximated more efficiently by relaxing some of the constraints on what the eigenvectors can look like. (The approximate eigensystem allows eigenvector elements to have real values rather than one of two specific values. After finding the right generalized eigenvector, we can threshold each of the values in the vector at some point to create a binary vector that approximates the actual binary vector achieved by doing the original NP-hard calculation.)

The algorithm boils down to the following steps:

1. Create a weighted graph from the pixels of the image, where the weight function is dependent on distance and other factors you might choose, such as color distance or brightness (see Eq. 4). Vertices more than a certain physical distance (r) apart are not connected at all in the graph.

$$w(i, j) = \begin{cases} e^{\frac{-\|F(i)-F(j)\|_2}{\sigma_F} + \frac{-\|X(i)-X(j)\|_2}{\sigma_X}} & \text{if } \|X(i) - X(j)\|_2 < r, \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

   F is a measure you choose, such as color distance or brightness; X is physical distance in the picture; σ is a constant.
2. Use the eigensystem in Eq. 5 to solve for the eigenvectors with the smallest eigenvalues (using the Lanczos method, which takes advantage of the sparsity of the matrices to make the computation more efficient). Pick the eigenvector with the second smallest eigenvalue.

$$(D - W)y = \lambda D y \tag{5}$$

   D is a diagonal matrix where $D(i, i)$ is the total weight of edges leaving vertex i; W is an adjacency matrix for all the pixels in the image.
3. Threshold this eigenvector at 0 into a binary indicator vector. You can also threshold at the median value.
4. Partition the pixels into two segments based on the corresponding values in the indicator vector. You can partition these two components recursively if you want more than two segments.

The running time of this algorithm is $O(mn)$ where m is the number of steps the Lanczos method takes to converge in calculating the eigenvector system, and n is the number of pixels in the graph. Our implementation uses a sparse matrix eigenvalue calculator from SciPy, which uses the Lanczos method but still runs very slowly (around half an hour per image). Our implementation also produces output segmentations which seem very likely to be wrong. However, because of time constraints, the exact bug has yet to be uncovered.

3 Datasets

We used two datasets to run our image segmentation algorithms on. The first dataset, created by the Weizmann Institute, contains 200 grayscale images with ground truth segmentations that were hand-drawn by 3 different human subjects. These images consist of two types– one object and two object images, as seen in Figure 1 and 2, and are available to us as downloadable pngs.

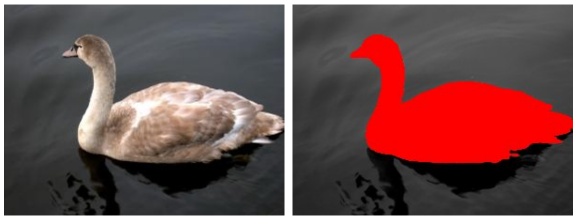

Fig. 1. A sample one-object image from the Weizmann dataset and one of the accompanying ground truth segmentations.

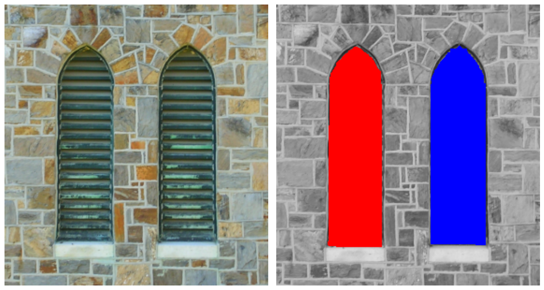

Fig. 2. A sample two-object image from the Weizmann dataset and one of the accompanying ground truth segmentations.

We decided to use the Weizmann dataset because the images are unambiguously segmentable, in that they contain only one or two objects, no more and no less. As a human, the 'objects' and the background of these images are easily observed, and so we could easily test our algorithms on these images. To start, we ran our algorithms on the simplest images, which were the Weizmann one object images. This allowed us to test if our algorithms performed as expected, to see if they succeeded in segmenting the image in relation to the ground truth, or if they were failing. After we were more confident that our algorithms performed successfully on the simplest images, we ran them on slightly more complex, two-object images.

The second dataset we used was the Berkeley Segmentation Dataset. The Berkeley dataset consists of 12,000 hand-labeled segmentations of over 1,000 images, segmentations created by 30 different people. A sample image and its segmentation can be seen in Figure 3. Three-hundred of these images in both grayscale and color are available for public use. The original images are available for download as jpgs, and the ground truth segmentations are available as .seg files, which essentially describe which pixels belong to which segments (the format is detailed on their webpage).

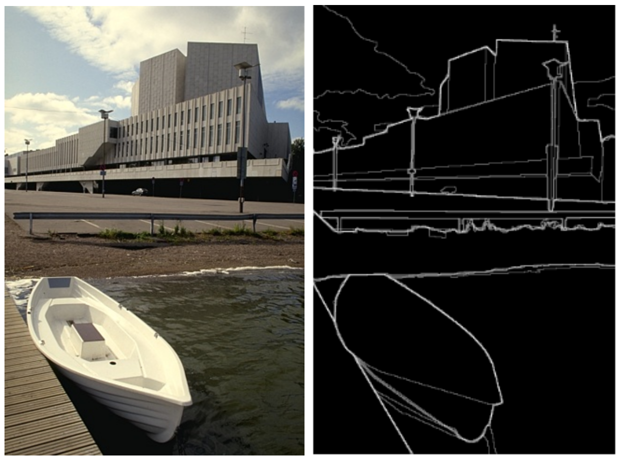

Fig. 3. A sample image from the Berkeley dataset and the accompanying ground truth segmentations (appearing here as an aggregate of all hand-drawn boundaries).

The Berkeley dataset provided us with more complex, real-world images with more complicated segmentations. Working with these types of images allowed us to evaluate what "simple" vs "complex" images were. Our findings will reveal how our algorithms performed on images that had a single object and were highly contrasted versus images that had a multitude of objects that could be segmented in many different ways. Findings will also show how different algorithms have varying ideas of what a simple or complex image is, as some perform better than others on what we deem as simple or complex.

Finally, we selected datasets not only for the types of images they contained and the ground truth segmentations, but also for their ease of access. These two datasets provided detailed information on how to download and use each dataset, as well as options to browse the dataset by image or by human subject to view the ground truth segmentations. They also provide several different human segmentations of each image.

It is also important to note that with both datasets, we ran our in-progress segmentation algorithms on a smaller subset of these images that we selected for testing purposes. For our data, we ran the algorithms on a larger subset of images, producing hundreds of data points each. All the algorithms segmented all images with the exception of Region Growing, where difficulties with runtimes prohibited us from running on every image in a dataset. Instead, Region Growing was tested on a smaller, random subset of images in each of the datasets.

4 Evaluation Metrics

To quantify the quality of our segmentations and better numerically compare the performance of our algorithms, we used two evaluation metrics. These two metrics attempt to capture different qualities of a segmentation: the Jaccard metric captures how well the regions of each segment match up in the ground truth image and the segmentation, while Boundary Displacement Error (BDE) captures how closely the edges of each segment match up. These two metrics were used occasionally in the literature, and we thought that they had the potential to capture important differences between our different segmentation techniques.

4.1 Region-Based Evaluation

The Jaccard metric is a metric for measuring how similar regions are to one another. It is computed in a straightforward manner. First, we calculate the overlap between the two regions, which is the number of pixels that appear in both regions. Second, we calculate the total number of pixels that appear in either region. We divide the overlap of the two regions by the total number of pixels in either region, and this yields the Jaccard metric for those two regions. Because the Jaccard metric increases as the two regions have more pixels in common and decreases as the two regions have more pixels not in common, it makes intuitive sense to use it to compare regions.

For our region-based evaluation metric, we iterate over every region in the ground truth image and find its highest-jaccard match in the generated segmentation. We repeat this process in reverse, finding the highest-jaccard matches for each region in the generated segmentation. Then, we average together all of these numbers to find the average Jaccard score of each region with its best match. By doing this, we are able to incorporate the Jaccard score into our region-based evaluation metric.

4.2 Boudary Displacement Error

Boundary Displacement Error (BDE) [8] is a measure of error that accounts for how far apart the edges of a segmentation and the edges in the ground truth are. It is conceivable that an algorithm might generally find the regions of the image correctly but get the shape of the objects in the image wrong. This would mean that the Jaccard score would be fairly good but the BDE score might not be as good. Thus we believed this metric would measure different qualities of a segmentation.

We calculate the BDE value by finding the minimum distances from each edge pixel in one image to any edge pixel in the other. We formalize this as the distance equation in Eq. 6: If B_1 and B_2 are the two sets of edge pixels (in the ground truth and segmentation respectively), and p_i is a pixel in B_1, we say the distance from p_i to B_2 is the minimum Euclidean distance from p_i to any pixel in B_2.

$$d(p_i, B_2) = \min_{p \in B_2} \parallel p_i - p \parallel \tag{6}$$

$$BDE(B_1, B_2) = \frac{1}{2}\left(\frac{1}{|B_1|}\sum_{i=1}^{|B_1|} d(p_i, B_2) + \frac{1}{|B_2|}\sum_{j=1}^{|B_2|} d(p_j, B_1)\right) \tag{7}$$

Overall, BDE is computed using Eq. 7. More intuitively, it is an average of how far B_1 is from B_2 and how far B_2 is from B_1. Thus a lower value of BDE will indicate a better match between the segmentation and the ground truth. (Two segmentations whose edges exactly matched up would have a BDE value of 0.)

It is important to compute the distance in both directions and have both terms of Eq. 7 in order to keep the metric balanced. For example, the ground truth in Fig. 4 includes the ground and the sky as separate segments, while the segmentation next to it does not. So there will be many edge pixels along the horizon that are not represented in the segmentation. If we only measure the distance from edge pixels in the segmentation to edge pixels in the ground truth, we will find a pixel that perfectly lines up for almost every edge pixel, and we will conclude we have a perfect segmentation with BDE close to 0. However, if we also measure distance from edge pixels in the ground truth to edge pixels in the segmentation, many of the edge pixels along the horizon will be quite far away from the closest edge pixel in the segmentation. This will result in a higher BDE value, which will better reflect the accuracy of the segmentation.

One major downfall of BDE is that, unlike our region-based evaluation metric, it does not measure a percentage of possible error or correctness. It is not immediately apparent what would be the theoretical upper bound on BDE score. So although we can theoretically interpret a BDE score as an average error distance for edge pixels, it can be a bit challenging to interpret the implications of that for any individual image. Therefore, when interpreting our results, we found it more difficult than we anticipated to interpret our BDE scores.

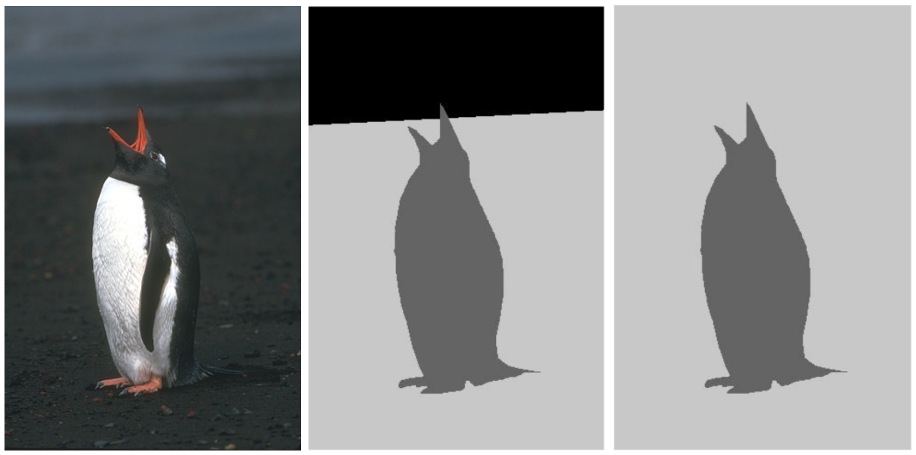

Fig. 4. Original image, ground truth, and a possible segmentation produced by an algorithm for an image from the Berkeley dataset.

5 Results

5.1 Region Growing

After running our implementation of region growing on hundreds of images, our segmentations showed that the algorithm performed best on high contrast images, and struggled with less contrasting ones where the object is similar in color to the background. This is reasonable since region growing depends on a threshold, a strict cut off variable. So, if the color of the object and the background are similar, they may fall below the threshold and be grouped, incorrectly, into the same region. However, on images that were highly contrasted, with an optimal seed and threshold, had high success rates.

Looking at the two metrics we used to evaluate our segmentation algorithms, the BDE scores for the Weizmann dataset proved that region growing produced accurate results according to the ground truth of the images. On average, for the Weizmann Dataset, the BDE score measured 9.1 and 3.8 for the one and two object images, respectively. For the Jaccard metric, average scores were 68% and 54% for the one and two object images. For the Berkeley Dataset, BDE was not calculated due to time, and Jaccard had an average success of 60%. .

Finally, an interesting discovery about region growing is the runtime. In the literature, region growing with multiple seeds runs in big-O time n*m*k, where n*m are the dimensions of the image and k is the number of seeds. A major drawback of our implementation is that it takes a long time to run, in other words, we did not get this runtime. Given an image with dimensions larger than 255 by 255, the segmentation may take as much as an hour or more to run. For this reason, images were resized into a 255 by 255 image, and though the output was compressed, it allowed us to visually analyze how this algorithm performed on different images in an acceptable time of a few seconds to a few minutes.

However, it is also important to note that this created issues down the road with the evaluation metrics, leading to fewer results as compared to the other algorithms.

5.2 Kmeans

Running kmeans across our datasets yielded relatively good results, particularly on images with with higher contrast and homogeneous segments. In cases such as Fig. 5 where the groundtruth segments in an image were each relatively consistent in color, and distinct from other segments, kmeans tended to do very well. In contrast, it struggled on images such as Fig. 6 with more within cluster variance, or ones were multiple clusters had similar colors. Since kmeans didn't take physical distance into account, it was particularly vulnerable to producing "speckled" segmentations, where individual pixels of one segment would be scattered throughout another segment, often caused by color variance such as flecks of sunlight.

Kmeans did well in choosing the number of segments via the elbow method. Very few images are dramatically under or over segmented, which greatly improves kmeans performance.

The median BDE scores for kmeans were 7.714 for Weizmann 1, 6.266 for Weizmann 2, and 8.676 for Berkeley.The median Jaccard scores for kmeans were 51.2% for Weizmann 1, 54.3% for Weizmann 2, and 9.1% for Berkeley. Visual inspection shows most images decently segmented, with the exception of "speckling" throughout.

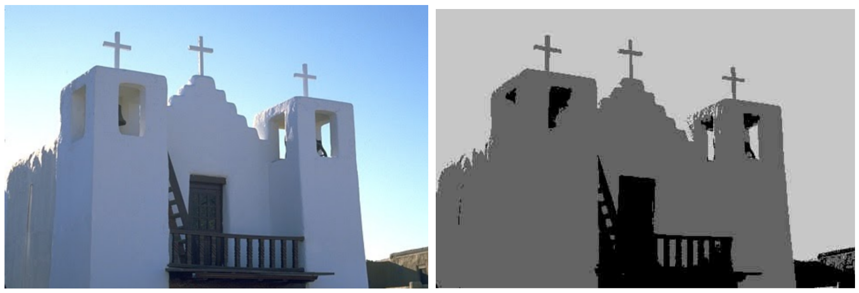

Fig. 5. Original image and segmentation from the Berkeley dataset.

5.3 Minimum Cut

Given the likely existence of a bug in the minimum cut algorithm, results obtained on our dataset images were hard to conclusively analyze. Segmentations obtained almost all had a very distinctive striped quality to them, which we attributed to the bug. This seemed to be somehow linked to the distance r beyond

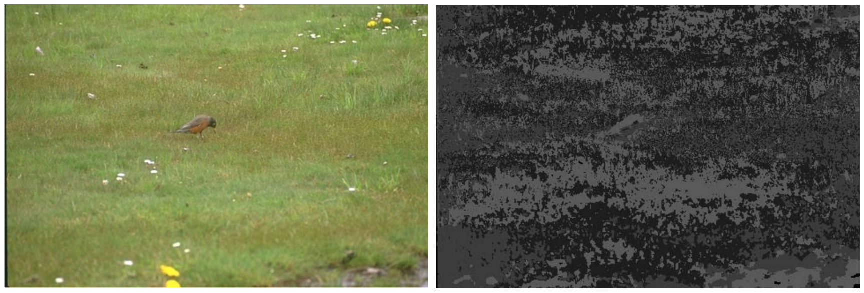

Fig. 6. Original image and segmentation from the Berkeley dataset

which pixels would not be connected in the graph (see Eq. 4), as segmentations created with different values of r had a clear difference in stripe width (see Fig. 7. Additionally, segmentations where the graph edge weight took into account only RGB distance had much less angular stripes than those that took into account both RGB distance and physical distance. Further testing is necessary to figure out why exactly these factors would be linked to the bug.

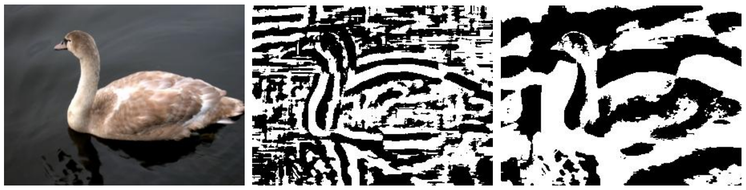

Fig. 7. Original image from the Weizmann dataset and min cut algorithm segmentations for $r = 16$ and $r = 30$.

For our experimentation, we ran the min cut algorithm with $r = 12$ using RGB color distance only in the edge weight. In general, this formation of the algorithm seemed to detect the edges of regions fairly well and fairly consistently. We see BDE scores tightly clustered around 10 for all datasets. Jaccard scores were also clustered very tightly for each dataset, and there is a clear decrease in Jaccard scores as the number of segments increases (Weizmann 1 Object scores are much higher than Weizmann 2 Object, which are much higher than Berkeley scores). This makes sense with the way that Jaccard score is calculated, since we only ran min cut to generate two segments, so the dataset with two segments in the ground truths matches more closely than those with more than two segments in the ground truths.

The average BDE score for min cut was 10.5 for Weizmann 1, 9.7 for Weizmann 2, and 10.5 for Berkeley. The average Jaccard score was 52% for Weizmann 1, 30% for Weizmann 2, and 4.6% for Berkeley. Upon visual inspection, most outlier points did not seem noticeably worse or better than the average segmentations, so it is unclear what exactly led them to be outliers.

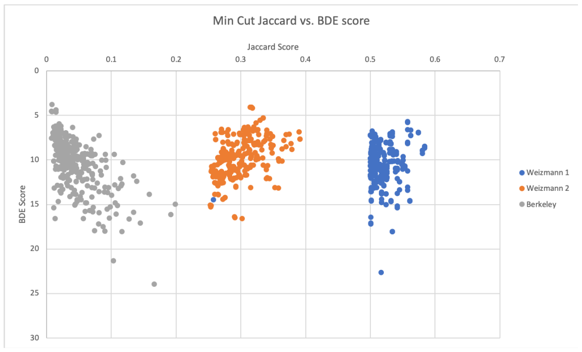

Fig. 8. Minimum cut scores for Jaccard score plotted against BDE scores for all three datasets.

5.4 Comparative Results

We ran our six algorithms on all datasets and evaluated their results with both metrics, with a few exceptions. Firstly, region growing was only run on a limited subsection of the images from all datasets, and is thus not included in the results tables. Secondly, thresholding was not run on the Weizmann 2 dataset, as it is designed to segment into precisely two segments, not the three that that dataset wants. Lastly, thresholding's results on the Berkeley dataset were only evaluated with Jaccard, not BDE. When scoring our segmentations for the Weizmann dataset, which included multiple ground truths for each image, we took only the best score, as there might possibly be multiple valid segmentations, one of which our algorithm finds, and we shouldn't penalize it for not being able to match multiple, potentially highly variable, ground truths.

When evaluated with the BDE metric, we notice relatively close results across our different algorithms. For each algorithm that was run on all three datasets, it scores best on the Weizmann 2 images. This is possible because the Weizmann images are simpler than the Berkeley ones, making them easier to segment, while the tendency of some algorithms to over segment makes them score worse on an image with only two ground truth segments. Overall, kmeans, split and merge, mincut, and thresholding perform very similarly, while watershed does a little worse. Mincut also has a much smaller distribution of scores across different images.

When evaluated with our Jaccard metric, we get very different results. Median scores range from 4% to 85% across different algorithms and datasets. Notably, watershed now performs the best across all datasets, with the exception

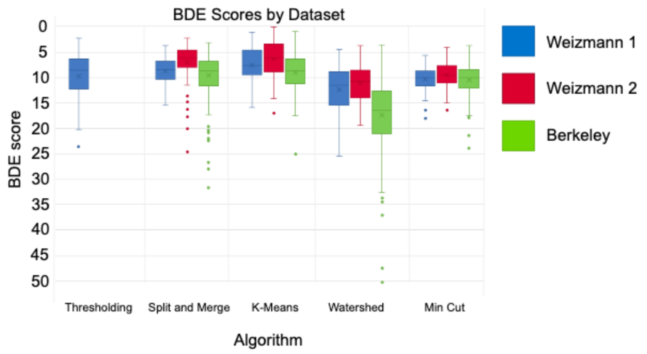

Fig. 9. BDE scores by dataset

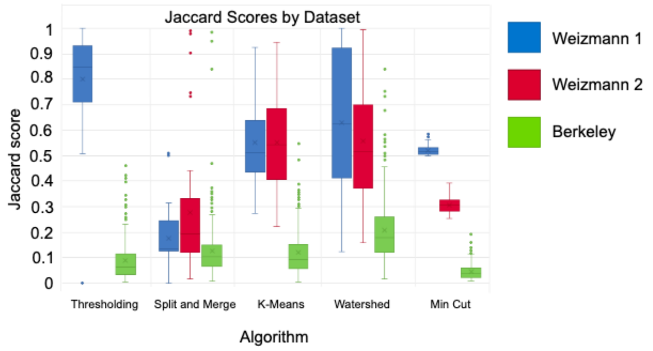

Fig. 10. Jaccard scores by dataset

of thresholding on Weizmann 1 (Understandably, as thresholding specializes in splitting an image into exactly two segments). Other noticeable changes from the BDE scores include Split and Merge's much worse performance, and the amount that mincut's scores change across datasets (though it still has very uniform results within a dataset). Evaluated based on region rather than edges, Weizmann 1 seems to be easier to segment for most algorithms, possibly due to region based evaluation penalizing over-segmentation less.

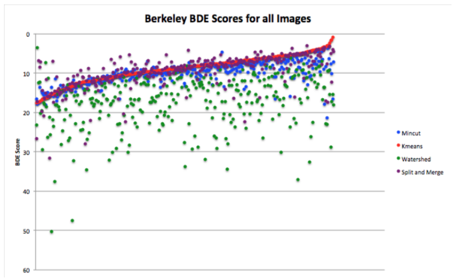

Fig. 11. Jaccard scores by dataset

Also interesting is to compare our algorithms on an image-per-image basis. When sorted by the BDE score on the kmeans algorithm, the graph for the berkeley dataset shows a noticeable trend in the two out of the other three algorithms matching with kmeans. Mincut and split and merge tend to do similarly to kmeans on the same images, while watershed does worse overall, and has more variance, but still trends in the same direction as the other three. This pattern is also present, to a lesser extent, in the Weizmann results when scored with BDE, but noticeable absent for all datasets when scored with Jaccard.

Lastly, because we saw trends in data varying significantly based on which evaluation metric was used, we looked at overall scores for both at once. Unfortunately, there doesn't seem to be any relation between which segmentations are considered good by the two metrics.

6 Conclusion

When comparing the ultimate performance of all of our algorithms, it can be quite hard to draw conclusions as to which algorithm performed the best. For example, in our comparison of BDE scores, it would be easy to conclude that within each margin of error each algorithm performs roughly the same, with each encountering additional difficulty with the Weizmann dataset the bulk of results remaining in a rough BDE score of 5-15. However, the results for the Jaccard metric are far more varied, which might lead one to believe that this metric would be the better gauge of accuracy for our metrics. From the BDE and Jaccard data, we can deduce that split and merge and min cut fare quite poorly

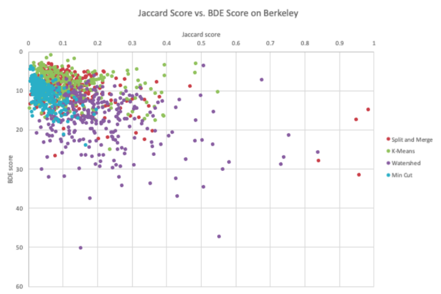

Fig. 12. Jaccard scores by dataset

on Jaccard, while watershed and threshold maintain their accuracy across both metrics. However, further examination shows that almost all of the algorithms fared poorly when evaluating their results on the Berkeley Dataset using Jaccard; a dataset that contains a far greater degree of natural images, not to also mention a far more robust means of creating ground truths. We also have to take into account the large distribution of values for all the metrics minus min cut. It would be difficult to justify that an algorithm works well if it only delivers its most accurate results for only a few select images.

While this data does not clearly define which algorithm is the best, we can draw a few conclusions from the data about these algorithms. For one, all of these algorithms perform decently to excellently at creating boundaries going off of the data from BDE, but only a few algorithms, namely thresholding, on certain datasets produce the full regions that are consistent with the ground truths, as indicated by the data in Jaccard. It can also be concluded that none of these metrics perform optimally on large, natural images like the ones found in the Berkeley dataset.

There are of course a few technical flaws as well. Our min cut and split and merge algorithms have not been completely implemented correctly, given time constraints, and this certainly will impact our results. We also do not have a good sense of how well region growing performed compared to the other algorithms, and that could change much of our perspective on these results. Given the data we do have, region growing appears to do quite well on all datasets, doing especially well on Berkeley in terms of Jaccard score compared to other algorithms. However, because of the limited tests we were able to run, this average may not be comparable to the data we have for our other algorithms. There should also be some acknowledgement granted to the difficulty of comparing BDE scores, given that unlike Jaccard, it is not immediately obvious what would be the theoretical maximum BDE score; this can make definitions of which BDE scores are poor simply a matter of comparison to other scores, not necessarily whether they produce an unacceptable or acceptable segmentation.

Future tests could take a variety of forms. One approach would be a combined test of algorithms, where we test algorithms run one after the other in order to enhance a segmentation. This research would be beneficial as it is possible that because these algorithms seem quite specialized, that perhaps what is necessary to make them more flexible is to use other algorithms to create a ”source” image (a basic segmentation) to enhance accuracy.

Additionally, many of our algorithms have multiple different parameters that can be tuned. While often we decided on fixed, random, or somewhat arbitrary values in the interest of time, experimenting with fine-tuning these parameters could make a big difference in the effectiveness of these algorithms in general, or in segmenting particular difficult images.

In summary, while some of these algorithms have clear strengths in the context of certain metrics, such as with thresholding and watershed on Jaccard scores, and most algorithms in BDE, there is not a conclusively “best” algorithm. The best answer is that having all of these algorithms and more to attempt on a given image would be the ideal approach, due to the high variance of scores on the variety of images and metrics employed. The process of image segmentation is a fascinating and difficult one, and it comes as little surprise that a varied toolkit is important to making it work.

References

1. Chauhan, N.S.: Understanding k-means clustering in machine learning (07 2019), https://towardsdatascience.com/introduction-to-image-segmentation-with-k-means-clustering-83fd0a9e2fc3
2. Dingding Liu, Bilge Soran, G.P.L.S.: A review of computer vision segmentation algorithms
3. Garbade, D.M.J.: Understanding k-means clustering in machine learning (09 2018), https://towardsdatascience.com/understanding-k-means-clustering-in-machine-learning-6a6e67336aa1
4. Krajíček, V.: Segmentation algorithms
5. Otsu, N.: A Threshold Selection Method from Gray-Level Histograms, vol. 9 (01 1979)
6. Shi, J., Malik, J.: Normalized cuts and image segmentation. IEEE Transactions on Pattern Analysis and Machine Intelligence **22**(8), 888–905 (2000)
7. Ville Satopa, Jeannie Albrecht, D.I.B.R.: Finding a "kneedle" in a haystack: Detecting knee points in system behavior
8. Wang, X.: Graph based approaches for image segmentation and object tracking (2015)
9. Xin Zheng, Qinyi Lei, R.Y.Y.G., Yin, Q.: Understanding k-means clustering in machine learning (08 2018), https://doi.org/10.1186/s13640-018-0309-3