Extending GrabCuts: Multi-Object Image Segmentation for Enhanced Feature Extraction

Viyath Fernando (viyathf2), Peter Florido (pflorid2),
Varun Gangadharan (varunpg2), Colleen Heinemann (heinmnn2)

May 13, 2024

1 Introduction

Image segmentation is a technique in the realm of computer vision that partitions an image into discrete groups of pixels, or segments the image, to detect objects/features/etc. [1] These techniques can be extremely simple or very complex and have numerous applications. There are instances where utilizing a technique such as graph cuts can be particularly useful by turning an image into a graph and splitting the nodes, or pixels in many instances, into objects within an image based on their neighboring nodes and characteristics. One particularly noteworthy example of this type of algorithm is the GrabCuts algorithm put forth by Rother et al. in 2004 [2]. In this algorithm, they provide an interactive iterative algorithm to extract features from the foreground of an image. For example, if there is a person standing in front of a forest, the goal would be to extract all pixels of the image pertaining to the person and leave the background. Figure 1 shows an example of this taken directly from the paper by Rother et al. and shows how the user can extract the features, or foreground, from the background of the image.

[1]https://www.ibm.com/topics/image-segmentation

[2]https://www.microsoft.com/en-us/research/wp-content/uploads/2004/08/siggraph04-grabcut.pdf

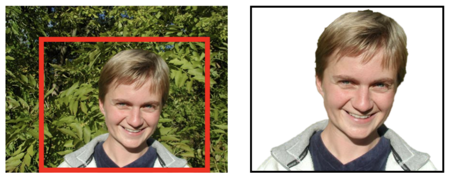


Figure 1: Example of GrabCuts algorithm being used to extract a person, or the foreground, from the leaves behind them, or the background

While GrabCuts can be an extremely useful algorithm, it is limited to extracting only one feature, or one foreground, from the background. What if there is more than one feature we want to extract? In this work, we extend the work of the GrabCuts algorithm by expanding the trimap to a quadmap and allow the user to interactively extract more than one feature, or foreground, from the image. We then analyze how well our algorithm performs to see if it is an accurate enough measure to use.

2 Algorithm

2.1 Two Label Problem

Image segmentation essentially boils down to a (0, 1) problem where the foreground gets labeled as 1 and the background gets labeled as 0. In the simplest case, we can use a trimap to find the optimal solution to an interactive segmentation problem, similar to what can be seen in Figure 1. The trimap has 3 labels – 0 is the background, 1 is the foreground, and X is unknown. With GrabCuts, when there is a part of the image that does not get chosen as the foreground, we can mark that part of the image to explicitly be a part of the foreground and include it with the rest of the foreground object.

The framework to do the above is as follows: minimize an energy that depends on pixel values and parameters and is a function of the labels. α_n is equal to 0 if n is the background and α_n is equal to 1 if n is the foreground. There are two terms. One term is a data term that is linear in α. The other term is a smoothness term that is quadratic in α. z is the pixel values, θ are a bunch of parameters that will talked about later. The optimization problem is: minimize for a given θ and z in α's. The equation is as follows:

$\boldsymbol{E}(\alpha,\, \theta,\, z = \boldsymbol{U}(\alpha,\, \theta,\, z) + \boldsymbol{V}(\alpha,\, z)$

We now look at the first term in the energy, or the linear term. It is the probability of obtaining the pixel value at location n, conditioned on whether it is foreground or background. To find the probability model we look at:

$\boldsymbol{U}(\alpha,\, \theta,\, z) = \sum_n -logh(z_n;\alpha_n)$

We want pixels to be like their neighbors. If the pixel and its neighbor have similar brightness, or about the same color, we want them to agree. m and n are neighbors on a grid and we will sum over the distance of m and n. If we're working with neighbors on a grid at the pixel level, the distance is always 1. Then we have a charge that asks if they have the same label, which is a part of the optimization problem. We also then have a weight that makes the cost smaller if the labels of the pixels m and n are very different. This will give a smaller smoothness cost for pixels that disagree if the pixel colors are very different. The equation for this is:

$\boldsymbol{V}(\alpha,\, z) = \gamma \sum_{(m,n\, \in C)} dis(m,n)^{-1}[\alpha_n \neq \alpha_m] exp{-}\beta(z_m \text{ - } z_n)$

We basically have two problems that we have to deal with here:

1. impute probability model from markup

2. given the model, compute the solution

The simplest way to go about this is in the GrabCuts paper by Rother et al., which is to build a trimap. With a trimap it is relatively straightforward to use the background and foreground example pixels to build a probability model. For each pixel, we can think about the data term as follows:

α_n[-log$p(z_n|foreground)$] + (1 - α_n)[-log$p(z_n|background)$]

If α is 1, then we get [-log$p(z_n|foreground)$]. If α is 0, then we get (1 - α_n)[-log$p(z_n|background)$]. If α is 1, we get the log probability of being in the foreground. If α is 0, we get the log probability of being in the background.

The smoothness term costs us something if α_n is not the same as α_m. We can write this as the sum of a clique, which is really a neighborhood in a grid graph, α_n(1 - α_m) plus (1 - α_n)α_m. If they disagree with each other, either α_n times one minus α_m or one minus α_n times α_m is equal to one. If we look at the quadratic term, we see it's better to agree than disagree because we're minimizing. The smoothness term is as follows:

(linear term in α - $\sum_{(m,n\, \in C)}(\alpha_n \alpha_m)[\gamma$exp(-$\beta$]$||z_m$ - $z_n||^2$)]

All of the pixels outside of the box will be labeled background. All of the pixels inside of the box will be labeled as foreground. We'll use all of the background pixels to build a background pixel probability model. We'll use all of the foreground pixels to build a foreground pixel probability model. All of the pixels inside of the box are unknown and we'll label them solving an optimization problem. We're going to iterate the estimate of the pixel probability model and the segmentation.

For every pixel we'll estimate $\boldsymbol{f}_n$ because we know whether it is a foreground or a background pixel. We will choose the normal with the largest likelihood for the normal of that pixel given if it's foreground or background. That is the normal distribution, $\boldsymbol{f}_n$. We now know all of the $\boldsymbol{f}_n$ and the α_n. For each pixel, we know the normal distribution it's associated with, so we re-estimate the mean and covariance of each normal distribution. We now estimate α_n for each pixel given the mean, the covariance, and the $\boldsymbol{f}$ and we do so by cutting the graph. We then iterate that because the foreground and background labeling may not be consistent with the mean, covariance, and $\boldsymbol{f}_n$. We re-estimate $\boldsymbol{f}_n$, the mean, and the covariance to re-estimate α_n. We do this until the energy does not decline.

2.2 Three Label Problem

This idea can be expanded beyond just finding one object in an image to two objects in an image. This leads to three labels instead of two and we now have to estimate α_n. Cutting the graph will no longer work like it did for a two label problem such as choosing one object in an image.

We will work on all of the vertices and the edges will be every directed edge except for the source and the target. Any edge not in the original graph gets a capacity of 0. A flow is a mapping from edges to the real numbers. It's feasible because the flow on the edge from v to w is less than or equal to the capacity from v to w, which is represented with:

$\boldsymbol{f}(v \to w) \leq \boldsymbol{c}(v \to w)$

We also require that the flow going from v to w is equal to minus the flow from w to v, which is represented with:

$\boldsymbol{f}(v \to w) = \text{-}\boldsymbol{f}(w \to v)$

This means there could be negative flows, but this isn't an issue because the capacity stops at 0 (capacity is always positive). We also require that all of the flow summed from u to v is 0 for all vertices, which is known as Kirchoff's law. Everything that comes in also goes out. This is shown with:

$\sum_u \boldsymbol{f}(u \to v) = 0$ for all $v \in V$ - $\{s, t\}$

We are able to sum everything that is leaving the vertex because of $\boldsymbol{f}(v \to w) = \text{-}\boldsymbol{f}(w \to v)$. If there is positive flow coming in, that means that there is negative flow going out. The value of the flow is the sum of everything going into the target:

$\sum_v \boldsymbol{f}(v \to t)$

The problem with flow is that it has complicated linear constraints. At every vertex, the stuff coming in must be equal to the stuff going out. We are going to relax this constraint and create a novel object called a pre-flow. A pre-flow is a map from the edge to the real where the flow of every edge is less than or equal to the capacity. The flow going from v to w is minus the flow from w to v. All edges leaving u added up is greater than or equal to 0. This then leaves the residual, which gets updated by subtracting the amount from the residual in both directions. The excess gets updated in both directions as well.

The second operation is relabeling. Relabeling is admissible if the vertex is active, which means it doesn't have an infinite label and it has excess. There's some residual leaving that vertex, so we are going to find the edge with the residual leaving that vertex, implying $d(v) \leq d(w)$. Then:

$d(v) = \min d(w) + 1$ over edges where $\boldsymbol{r}_f(v \to w) > 0$

If $\boldsymbol{f}$ is a pre-flow, d is a valid labeling, and v is an active vertex, then either a push or a relabel is admissible for v. If $\boldsymbol{f}$ start a pre-flow and v is a valid labeling, they will remain a pre-flow or a valid labeling, which is known as induction.

If $\boldsymbol{f}$ is a pre-flow, and d is a valid labeling on V, then we can create a new graph that takes the vertices of the original graph and all of the edges that have residual capacity, but not the other edges. In the new graph, you cannot get from the source to the target. There is not a sequence of edges with residual capacity that goes from the source to the target. An augmented graph would be when residual is pushed along the path from the source to the target, but in this instance there is no path that will go from the source to the target. To prove this, we assume there is a path from s to t, which is the source and the target, respectively. The path consists of edges that have residual capacity. We also have a valid labeling where $d(v_i) \leq d(v_{i+1}) + 1$.

The label of v_0 is the label of the start in this path is less than or equal to the label in the target plus m and all of this is less than $|V|$. This means that:

$d(v_0 = d(s) \leq d(t) + m < |V|$ because $m \leq |V|$ - 1 and $d(t) = 0$

but $d(s) = |V|$ for valid labelling, which is a contradiction, so the augmenting path does not exist.

If the algorithm terminates and all labels are finite, then $\boldsymbol{f}$ is a flow and is also maximal. At termination, there is no admissible vertex. If there was, we'd be able to do something. This means it is a flow because it satisfies Kirchoff's law. There is no augmented path, as we just proved, so it has to be maximal.

In the case of our work here, we want to look at three labels such that r = 3. This will reduce to a linear program in the same way that the other one did, but that linear program does not have total uni-modularity and you're not going to get them because it's not a polynomial linear program. We first need to turn it into a linear program. If there is some kind of probability model per class and we use some type of smoothing where it's expensive to be different from the neighboring pixel, then we are going to minimize the linear term in l_{ij} plus a quadratic term in l_{ij}. The constraints become:

$l_{ij}{}^T 1 = 1$

$l_{ij} \in \{0,1\}^r$

This is easily turned into a linear program. For every location, we come up with a one half vector. This is the vector above that is all 0's except for one 1 somewhere in the vector. The singular 1 is in the location of the label that we are interested in. That vector at this location must satisfy the properties listed above including that the label vector dot the vector of all 1's is equal to 1 as well as the label vector is a 0, 1 vector. For r = 3, there will be a probability of observing the value of the pixel given foreground1, the probability of observing the value of the pixel given foreground2, and the probability of observing the value of the pixel given the background. Doing this will end up with the same class of problem where we have a linear term in the label vectors, which will be the first element of the label vector times the probability of the pixel color conditioned on coming from first plus the second element of the label vector times the probability of the pixel color conditioned on coming from second plus the third element of the label vector times the probability of the pixel color conditioned on coming from third. Two of these will be multiplied by 0 and one will be multiplied by 1.

3 Analysis Metrics

In order to determine whether or not our implementation of GrabCuts for two foreground objects was correct, we needed to figure out how accurate our output was compared to the original GrabCuts algorithm that works to extract one foreground object. To do this, we started by first re-implementing the GrabCut algorithm from Rother et al.

To ensure that our GrabCuts implementation was working, we implemented an accuracy measure to analyze our output. Our calculation is comprised of taking the segmentation mask from our input image and dividing the foreground area by the total area. We then chose to compare the accuracy of using a trimap

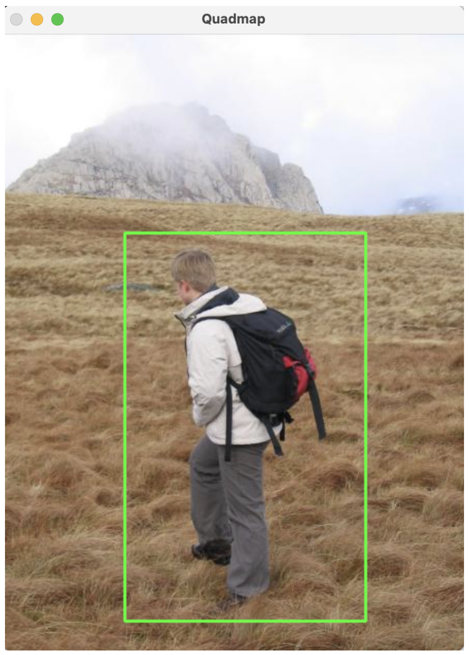


Figure 2: Example of allowing the user to draw a box around the feature that they want to extract from the given image

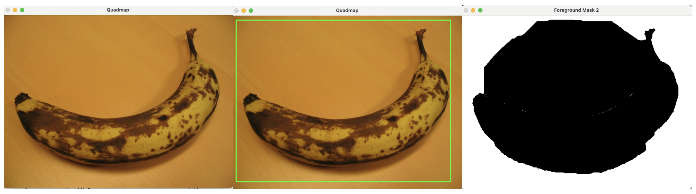
and our re-implementation of the GrabCuts algorithm to the accuracy of using a quadmap and our updated version of the GrabCuts algorithm. To test the robustness of our implementation, we ran on a variety of images, all greater than 600 pixels by 400 pixels in size.

4 Analysis

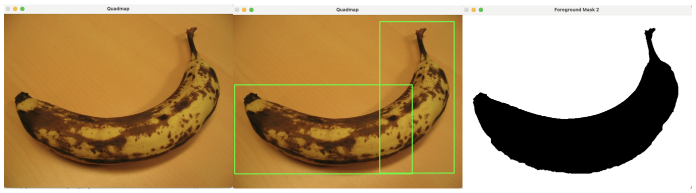
4.1 Interactivity

Our original re-implementation of the original GrabCuts algorithm did not give the user the option to interact with the images and draw their own boxes around the features to which extract. Instead, our original version simply defined a box at a given coordinate in an image as well as the size of that box. The problem was that, depending on the image and the size of the given image, this box could be exactly around the feature to detect or it could not even include the feature. The variability of this approach led to wildly inaccurate results for some images and relatively accurate results for others.

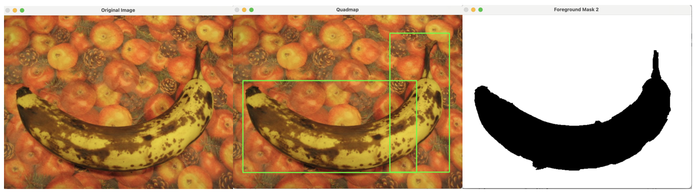
To bypass this issue, we chose to implement the ability for the user to interact with the image and draw their own box around the feature that they want to extract. Figure 2 shows an example of what this looks like to draw a box around one feature of an image.



(a) (b) (c)



(d) (e) (f)



(g) (h) (i)

Figure 3: (a) Original image; (b) Original image with one box drawn by user; (c) Output of using one foreground object; (d) Original image; (e) Original image with two boxes drawn by user (f) Output of using two foreground objects; (g) Original image with background noise; (h) Original image with background noise with two boxes drawn by user; (i) Output of using two foreground objects for image with background noise

We found that allowing the user to interact with their images saved a lot of trial and error in trying to find the correct location and size for the box, but also provided far more accuracy and gave much better output compared to hard coding the box.

4.2 Accuracy

Simply by looking at the output by eye, there are instances where it is evident that using our expanded GrabCuts algorithm to choose two foreground objects is much more accurate than choosing just one foreground object. Figure 3 is an example where using two foreground objects is much better than using one foreground object. We can see in Figure 3(c) that using one foreground object did not work well with the given input image; however, in Figure 3(f) we used two foreground images to break up the object and more accurately outline it.

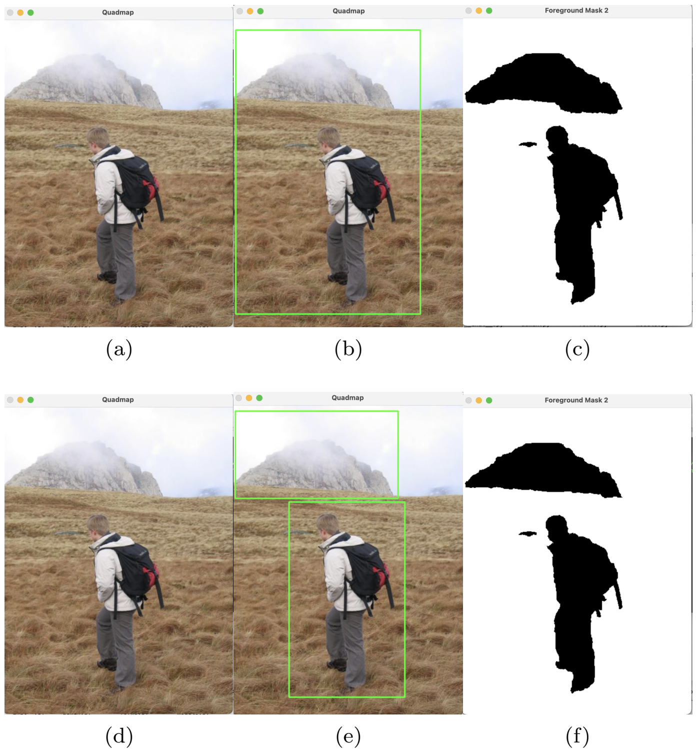


Figure 4: (a) Original image; (b) Original image with one box drawn by user; (c) Output of using one foreground object; (d) Original image; (e) Original image with two boxes drawn by user (f) Output of using two foreground objects

This led to much more accurate results visually. These visual results were backed up by our experiments where we tested the accuracy of our implementation and these values are included in Table 1. When running with one object, the output covered a significant portion of the image, as can be seen by an accuracy of .7922. When running with two objects, though, we were able to more accurately create a mask of the object. This gave us an accuracy of .2244 for one foreground object and an accuracy of .2978 for the second foreground object, for a total of .5522, which seems more appropriate.

We found instances where, by visual inspection, using one foreground object produced very similar results to using two foreground objects. An example of this is in Figure 4. Figure 4(c) shows the output of using just one foreground object while Figure 4(f) shows the results of using two foreground objects. These outputs are very similar, even including a small artifact next to the person's head that should not be included; it is included in both instances.

Through our experiments, we found that having background noise in an image, such as a decorative backdrop, trying to extract a singular tree out of

	Foreground1	Foreground2
banana$_bkgd_1object$	.7922	X
banana$_bkgd_2object$	.2244	.2978
mountain$_person_1object$	.6952	X
mountain$_person_2object$	.3357	.4299
$2_people_1object$	.8854	X
$2_people_2object$	.3398	.3541

Table 1: Caption

a forest background, etc., provides less accurate results. How inaccurate the results are appears to depend wildly on what the foreground object is that we wish to extract combined with what the background is comprised. As can be seen in Figure 3(g), (h), and (i), having a more complex background led to extra artifacts on the final foreground object, even when using two foreground objects to extract features from the image.

In order to test for accuracy, we chose images that could clearly be split into two foreground objects. We then ran these images by interactively choosing one foreground object (which included the two objects we are interested in). An example of this is in Figure 4 where we first drew one box to include the person and the mountain. Then we drew two boxes–one around the person and one around the mountain. The accuracy results are in Table 1. In this instance, Foreground1 is the mountain and Foreground2 is the person. As we can see both from the numbers and from a visual inspection, the mask for the person came out extremely similar, with .6952 being the how much of the image was covered by the mask using one object. Using two objects led to an accuracy of .3357 for the mountain and .4299 for the person, totaling .7656. When we look at the images, we can see that we are missing part of the mountain when only using one object. When using two objects, though, the bottom portion of the mountain is filled in as would be expected. In this instance, using two masks was more accurate than using just one.

Out of 30 images we tested with, 12 worked significantly better with two foreground objects than one. For example, when running with an image with two people standing next to each other, using only one object produced results that included a large portion of the background as the foreground object. The accuracy for this is included in Table 1 and was .8854, which means the output mask covered a significant portion of the image. When running with two objects, though, the results were much more reasonable with accuracy values of .3398 for one person and .3541 for the other person, leading to a total of .6939. This difference in values between using one object and two objects in this instance is due to the fact that the space between the two people was included when using one object, but was omitted when using two objects.

Our experiments showed us that there is a great deal of variability when it comes to accurately extracting features from an image. So many variables can come into play – the size of the object to extract, the number of objects to extract, the complexity of the background, and the combination of all of the above, for example. We found through our testing that, as is probably expected, it is much easier to extract a book from a blank table than to extract a single tree from a forest of trees. This means that there is a much more nuanced approach to accurately extracting a feature from an image than to simply say a given algorithm is good or not. An algorithm may be good for extracting a given feature from a given image, but may not be good at extracting a different feature from a different image. This means there is a much more complex approach to image segmentation and feature extract than simply saying ”use this algorithm” or ”don’t use this algorithm”.

4.3 Branch and Bounding

Our approach to extending the GrabCuts algorithm to work with two foreground features as opposed to one was to use alpha-expansion and extend the GrabCuts trimap into a quadmap. This was to allow for more than two labels (or one foreground and one background) at a time. Based on the information we researched and lectures from class, utilizing the branch and bounding method would not have been a feasible approach to our solution given the size of the images that we chose to test with. Most images we tested with were greater than 600 pixels by 400 pixels, meaning that branch and bounding would not have worked given that it is only viable when images are smaller than approximately 40 pixels by 50 pixels.

This approach would not work well for the larger images that we chose to utilize due to the fact that the search space available would dramatically increase in size and, therefore, take drastically longer to run. It is possible that it would give a more accurate result, but at a much larger cost than our current approach.

5 Conclusion

The intent of this work was to expand the original GrabCuts algorithm by Rother et al. in 2004 to be able to extract more than one foreground object from an image simultaneously. We did so by taking the trimap that the algorithm was originally built on and extending it to be a quadmap and therefore, contain the ability to extract two foreground objects. To ensure that our algorithm was working, we run an accuracy measure to compare the results of using one foreground object to using our algorithm that uses two foreground objects. We found that image segmentation is much more complicated than simply saying that an algorithm works or doesn’t work. There are so many variables that factor into whether or not a foreground object was able to be detected successfully. As visual data becomes more readily available as well as more complex, we need ways to analyze these images in a way that is ensured to be accurate and thorough. While there is still a long way to go, we believe that expanding the GrabCuts algorithm allowed us to take a step towards more precisely and accurately detecting objects in images of varying degrees of complexity.