GrabCut Algorithm and Repetition

RexRusk

1 Introduction

GrabCut is an interactive image segmentation technique that utilizes user input, such as a bounding box and foreground or background brush, to separate the foreground of an image from the background.

It builds upon previous approaches for segmentation and matting, including Magic Wand, Intelligent Scissors, Graph Cut, Level Sets, Bayes Matting, and Knockout. By integrating concepts from graph cuts and border matting, GrabCut enables users to iteratively refine the segmentation result, leading to more accurate and user-guided object extraction from images.

GrabCut is an improvement over five existing segmentation techniques: Magic Wand, Intelligent Scissors, Graph Cut, Level Sets, Bayes Matting, and Knockout. Each of these approaches has its strengths and limitations, with some having issues with considerable overlap of foreground and background or difficulties in handling textured regions. Others require more user interaction or may not be suitable for certain situations, such as camouflage segmentation.

Magic Wand	Considerable overlap of foreground and background
Intelligent Scissors	For highly or un-textured regions, many alternative “minimal” paths exist.
Bayes matting	Introduced a trimap, when Tu is large, it needs more user interaction
Knockout 2	Similar to Bayes matting, with less quality
Graph Cut	Segment even in camouflage, with a trimap and max-flow algorithm
Level sets	Use energy minimization tool not as good as Graph Cut

Deficiencies of before 6 image segmentation algorithms

Comparison of GraphCut and GrabCut:

GraphCut employs gray values, GrabCut utilizes a Gaussian Mixture Model (GMM) in place of histograms to represent color space.

GraphCut performs a one-time cost cut, whereas GrabCut adopts an iterative energy minimization process, allowing for more precise segmentation.

GraphCut requires a set of initial seeds, while GrabCut can work with incomplete labelling derived from a bounding box.

Improvements in GrabCut:

GrabCut replaces the monochrome image model used in some previous approaches with a more advanced Gaussian Mixture Model, enabling better color representation.

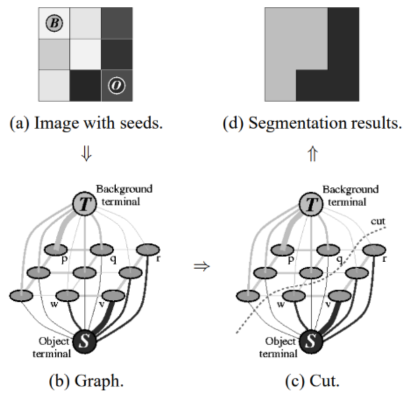
GrabCut replaces the one-shot minimum cut estimation algorithm with a more powerful iterative procedure, alternating between estimation and parameter learning.

GrabCut allows users to tackle challenging situations by employing foreground and background brushes for rough image segmentation through user interaction.

2 Algorithm Understanding

2.1 GraphCut Foundations

GrabCut is an image segmentation algorithm that leverages the concept of minimum cut in graph theory. It represents the image segmentation problem as a graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$, where $\mathcal{V}$ denotes the set of vertices (pixels in the image), and $\mathcal{E}$ represents the set of edges (neighboring pixels) connecting these vertices. These edges are often referred to as n-links, representing spatial relationships.



Each pixel in the image corresponds to a vertex in the graph, and neighboring pixels are connected by edges. To initialize the graph for segmentation, two special vertices are introduced: S (source) and T (sink). These vertices are initially fully connected to all other vertices in the graph.

The first step in the GrabCut algorithm is to obtain an initial trimap, denoted as T, which guides the segmentation process. The trimap separates the image into three regions:

Background pixels (T_B): Pixels outside the bounding box or initial selection, labeled as $\alpha_n = 0$ (background).

Foreground Pixels (T_F): Pixels inside the bounding box or initial selection that are considered definite foreground pixels. These pixels should be labeled with $\alpha_n = 1$, indicating that they are considered definite foreground pixels.

Unknown pixels (T_U): Pixels inside the bounding box or initial selection, labeled as $\alpha_n = 1$ (possible foreground).

The aim of GrabCut is to segment the image into foreground and background regions based on the given trimap. To achieve this, the algorithm interprets the original grayscale image $\mathrm{z} = (\mathrm{z1}, \mathrm{z2}, \ldots, \mathrm{zn})$ as a composition of a foreground image with varying transparency $\alpha = (\alpha_1, \alpha_2, \cdots \alpha_n)$ superimposed on a background image. The goal is to find the optimal transparency values α for each pixel, which yields the best segmentation results.

To determine the optimal transparency values, an energy function $E(\alpha, \theta, z)$ is defined, which consists of two terms:

Data term $U(\alpha, \theta, z)$: Represents the internal consistency of the segmentation based on the grayscale histogram of the pixels.

Smoothness term $V(\alpha, z)$: Encourages a smooth transition between adjacent pixels in the segmentation.

The segmentation quality is measured by minimizing the total energy function $E(\alpha, \theta, z)$ using the minimum cut algorithm. By applying the minimum cut, the algorithm finds the optimal partition between foreground and background regions, which corresponds to the best segmentation result.

2.2 GrabCut Improvements

The user inputs a rectangle that includes the object of interest in the image. Everything outside this rectangle is considered sure background (T_B). Everything inside the rectangle is labeled as unknown (T_U). Any user-provided hard labels for foreground and background regions are considered fixed and will not change during the segmentation process.

Gaussian Mixture Model (GMM) is used to model the foreground and background pixel distributions. It learns and creates new pixel distributions based on the given hard labels. The unknown pixels in the image are labeled either probable foreground or probable background, depending on their color relationship with the hard-labeled pixels. The GMM helps in capturing the color information and relationships in the image for segmentation.

Based on the pixel distributions obtained from the GMM, a graph is built. Each pixel becomes a node in the graph, and two additional nodes, the Source and Sink nodes, are added. Every foreground pixel is connected to the Source node, and every background pixel is connected to the Sink node. The weights of the edges connecting the pixels to the Source/Sink nodes are determined by the probability of a pixel being foreground/background, respectively. The weights between the pixels are defined based on edge information or pixel similarity, with lower weights for large color differences.

The segmentation process involves finding the minimum cut in the graph, which separates the Source node and Sink node with the least total cost. The cost function is the sum of weights of the edges that are cut. After the cut, all pixels connected to the Source node become part of the foreground, and those connected to the Sink node become part of the background. This process continues iteratively until the segmentation converges.

To achieve more natural edge transitions, the GrabCut algorithm introduces soft segmentation. It involves considering the transparency (alpha) of the pixels, allowing gradual transitions at the edges. The algorithm uses an energy function based on color differences and edge smoothness to determine the optimal alpha values. To avoid color bleeding along foreground edges, the algorithm borrows color information from the closest foreground pixel, and dynamic programming is used to find suitable alpha values.

3 Algorithm Outlines

My python script implements two main algorithms: GrabCut Segmentation and Border Matting.

GrabCut Segmentation Algorithm:

1. initializes the image, mask, and rectangle of bounding box and other parameters for foreground and background labeling.

2. Assign GMM components for foreground and background respectively.

3. Estimate GMM with K components and assign $\alpha_n = 0 \ or \ 1$ with EM algorithm.

4. Calculate the edges and weights of each vertex to estimate the min cut segmentation.

5. Calculate the energy function iterative from step 2 to 5 until convergence.

6. Estimate the mask from the min cut.

Border Matting Algorithm:

The main steps of the Border Matting algorithm involve finding the contour, grouping pixels, minimizing the energy function to find delta and sigma pairs, and constructing the alpha map (representing the opacity of each pixel).

1. Use an erode and canny algorithm to find the contour on the given trimap in the border pixels of T_F and T_B.

2. Find and calculate the minimum Euclidean distance the near contour pixel for each pixel in the trimap.

3. Minimize energy function with dynamic programming and obtain the best delta and sigma from smoothing regularizer traversal for each pixel on the contour.

4. Construct the alpha map with [0, 1] partition and multiply this map to the foreground image.

4 Math Understanding and Code Realization

First, they obtain a "hard" segmentation using iterative graph cut. This is followed by border matting in which alpha values are computed in a narrow strip around the hard segmentation boundary. Finally, full transparency, other than at the border, is not dealt with by GrabCut. It could be achieved however using the matting and this works well in areas that are sufficiently free of camouflage.

Energy Function

The image is now taken to consist of pixels z_n in colour space. As it is impractical to construct adequate colour space histograms, we follow a practice that is already used for soft segmentation.

The energy function is the cost associated with assigning a pixel to the foreground or background. The goal is to obtain the best possible segmentation. There are two terms: Data term and Smoothness term. Mathematically, the pairwise model can be expressed as E(labeling) = Data term + Smoothness term like this:

$$\mathbf{E}(\underline{\alpha}, \mathbf{k}, \underline{\theta}, \mathbf{z}) = U(\underline{\alpha}, \mathbf{k}, \underline{\theta}, \mathbf{z}) + V(\underline{\alpha}, \mathbf{z})$$

The Data term is related to the likelihood of each pixel belonging to the foreground or background, and it is usually based on color or texture features. The Smoothness term is influenced by beta and is used to measure the similarity between neighboring pixels. The data term U can be written as:

$$U(\underline{\alpha}, \mathbf{k}, \underline{\theta}, \mathbf{z}) = \sum_{n} D(\alpha_n, k_n, \underline{\theta}, z_n)$$

Where $D(\alpha_n, k_n, \underline{\theta}, z_n) = -\log p(z_n \mid \alpha_n, k_n, \underline{\theta}) - \log \pi(\alpha_n, k_n)$ consists of Gaussian probability distribution and mixture weighting coefficients, it can be written as:

$$D(\alpha_n, k_n, \underline{\theta}, z_n) = -\log \pi(\alpha_n, k_n) + \frac{1}{2}\log \det \Sigma(\alpha_n, k_n)$$
$$+\frac{1}{2}[z_n - \mu(\alpha_n, k_n)]^{\top} \Sigma(\alpha_n, k_n)^{-1} [z_n - \mu(\alpha_n, k_n)]$$
$$\underline{\theta} = \{\pi(\alpha, k), \mu(\alpha, k), \Sigma(\alpha, k), \alpha = 0,1, k = 1 \ldots K\}$$

The parameter β is determined by the contrast of the image, if the contrast of the image is low, that is to say, there is a slight difference between the pixels m and n, their Euclidean distance $\|Z_m - Z_n\|$ is relatively low, then we need to multiply by a larger β to magnify the difference, and vice versa. V term can work properly with either high or low contrast.

$$V(\underline{\alpha},\mathbf{z}) = \gamma \sum_{(m,n)\in\mathbf{C}} [\alpha_n \neq \alpha_m] \exp - \beta \|z_m - z_n\|^2$$

```
def calculateWeights(Zm, Zn, beta, gamma, diag=False):
    weight = beta * np.sum((Zm - Zn) * (Zm - Zn), axis=2)
    weight = gamma * np.exp(-weight)
    if diag: weight = weight / np.sqrt(2)# alpha n != alpha m
    return weight.flatten()  # flatten into a 1D array
```

The constant β iterates through the image pixels and calculates the squared Euclidean distance of neighbouring pixels. When $\beta = 0$, the smoothness term is simply the well-known Ising prior. Set $\beta > 0$ as this relaxes the tendency to smoothness in regions of high contrast. The constant β is chosen to be:

$$\beta = \left(2\langle(z_m - z_n)^2\rangle\right)^{-1}$$

To calculate the constant β from graph, I sum up the squared color differences for all neighboring pixel pairs and then use the edge number formula and the formula above to calculate β, the code can be written as:

```
def calculateBeta(img):
    rows, cols, _ = img.shape
    beta = 0
    for y in range(rows):
        for x in range(cols):
            color = img[y, x]
            if x > 0:  # left
                diff = color - img[y, x - 1]
                beta += np.dot(diff, diff)
            if y > 0 and x > 0:  # upleft
                diff = color - img[y - 1, x - 1]
                beta += np.dot(diff, diff)
            if y > 0:  # up
                diff = color - img[y - 1, x]
                beta += np.dot(diff, diff)
            if y > 0 and x < cols - 1:  # upright
                diff = color - img[y - 1, x + 1]
                beta += np.dot(diff, diff)

    if beta <= np.finfo(float).eps:
        beta = 0
    else:
        beta = 1 / (2 * beta / (4 * cols * rows - 3 * cols - 3 * rows + 2))
    return beta
```

Iterative GraphCut

First, the monochrome image model is replaced for colour by a Gaussian Mixture Model (GMM) in place of histograms.

In this function, I use RGB color space to model the target and background with a full covariance GMM (mixed Gaussian model). In grabcut function, input parameters are the raw image, mask image, bounding box, GMM components (typically $K = 5$), and the iterations of 3 in default.

```
def grabcut(img, mask, bbox, gmm_comps=5, num_iters=3):
```

In mask initialization, create a mask of the same size as the image. The mask is updated in each iteration to refine the segmentation of the foreground and background regions of the input image. In the initialization of the mask, which typically contains a rough estimation of the foreground and background regions. The function starts with parameter initialization and training of GMMs for foreground and background.

```
mask: ndarray = np.zeros(img.shape[:2], np.uint8)
```

The value of each pixel point in the mask is noted as 0 first. I set background, foreground, probable background, and probable foreground masks to 0, 1, 2 and 3 to represent it in the list index in the mask. Next, we determine the position of the rectangle we are drawing and set the bounding box in the portion of the mask to the probable foreground.

```
if np.count_nonzero(mask) == 0:
    mask[bbox[1]:bbox[1] + bbox[3], bbox[0]:bbox[0] + bbox[2]] = 3
```

Next, the GMM model is initialized by iterating over all the pixels of the entire image, taking all the pixels that could be background as sample pixels for the background, and all the pixels that could be foreground as sample pixels for the foreground.

```
bgd_gmm = GaussianMixture(n_components=gmm_components, 
covariance_type='full')
fgd_gmm = GaussianMixture(n_components=gmm_components, 
covariance_type='full')
```

Second, in each iteration, estimates the model parameters α_n with the EM algorithm using fit function in sklearn. This line trains a GMM to model the color distribution of the pixels in the background region. It takes the color data of background and foreground pixels and estimates the GMM parameters mean, covariance, and mixing coefficients based on the data they provided.

```
bgd_gmm.fit(img[back_idx])
fgd_gmm.fit(img[fore_idx])
```

Third, find weights for edges in foreground and background GMM prediction. After assigning to each pixel the Gaussian model to which it belongs in the GMM, the pixel weight is assigned to the Gaussian model with the highest probability in the foreground or background GMM.

In general, γ was set to 50. Lambda sets to 9 * γ Follows the OpenCV codes:

```
weights = np.hstack((weights, -bgd_gmm.score_samples(img.reshape(-1, 
3)[prob_idx])))
weights = np.hstack((weights, -fgd_gmm.score_samples(img.reshape(-1, 
3)[prob_idx])))
weights = np.hstack((weights, np.zeros(back_idx[0].size)))
weights = np.hstack((weights, np.ones(back_idx[0].size) * 9 * gamma))
weights = np.hstack((weights, np.zeros(fore_idx[0].size)))
weights = np.hstack((weights, np.ones(fore_idx[0].size) * 9 * gamma))
weights = weights.tolist()
```

Fourth, use the min cut to estimate the segmentation for each vertex in the image. Then the raw mask will be generated, it can be shown as:

```
# Perform GraphCut to save to the mask
graph = ig.Graph(2 + cols * rows)
graph.add_edges(edges)
```

```
pr_indexes = np.where(np.logical_or(mask == 2, mask == 3))

# 3. Estimate segmentation: use min cut to solve
mincut = graph.st_mincut(s, t, weights)
```
Finally, update the result to the foreground mask in each iteration:

```
# Update the mask based on the GraphCut result
mask[pr_indexes] = np.where(np.isin(idx[pr_indexes], mincut.partition[0]), 3,
2)
```
Don't forget to calculate the energy function until convergence:

```
# Calculate and store the energy for this iteration
energy = self.calculateEnergy(mask, weights)
energy_list.append(energy)
```

Border Matting

Using OpenCV Grabcut effect and thesis effect is quite different, in fact, OpenCV GrabCut effect and graph cut effect are quite the same. The reason is that OpenCV does not realize the border matting processing.

This algorithm is a method to estimate an alpha map for the strip without generating artefacts, and recovering the foreground colour, free of colour bleeding from the background.

The realization of the energy function is composed of the sum of data term and the sum of smoothing regularizer. For the reason that border matting is not the central algorithm in this paper, I only provide the annotated codes and my repository link in the appendix pages.

$$E = \sum_{n \in T_U} \tilde{D}_n(\alpha_n) + \sum_{t=1}^{T} \tilde{V}(\Delta_t, \sigma_t, \Delta_{t+1}, \sigma_{t+1})$$

$$\tilde{V}(\Delta, \sigma, \Delta', \sigma') = \lambda_1(\Delta - \Delta')^2 + \lambda_2(\sigma - \sigma')^2$$

$$\tilde{D}_n(\alpha_n) = -\log \mathbf{N}\left(z_n; \mu_{t(n)}(\alpha_n), \Sigma_{t(n)}(\alpha_n)\right)$$

$$\mu_t(\alpha) = (1-\alpha)\mu_t(0) + \alpha\mu_t(1)$$
$$\Sigma_t(\alpha) = (1-\alpha)^2\Sigma_t(0) + \alpha^2\Sigma_t(1)$$

1. Use an erode and canny algorithm to find the contour on the given trimap in the border pixels of T_F and T_B.

```
self.trimap = cv.erode(self.trimap, kernel=np.ones((3, 3), np.uint8),
iterations=2)
edges = cv.Canny(self.trimap, threshold1=2, threshold2=3)
```
2. Find and calculate the minimum Euclidean distance the near contour pixel for each pixel in the trimap.

```
m, n = self.trimap.shape
# Calculate the minimum Euclidean distance
for i in range(m):
    for j in range(n):
        min_dist = 100000000
        min_point = None
        for point in self.C:
            dist = (i - point[0]) ** 2 + (j - point[1]) ** 2
            if dist < min_dist:
                min_dist = dist
                min_point = point
        if min_dist < self.w ** 2:
            self.D[min_point].append((i, j))
```
3. Minimize energy function with dynamic programming and obtain the best delta and sigma from smoothing regularizer traversal for each pixel on the contour.

```
for delta in range(1, self.delta_level):
    for sigma in range(1, self.sigma_level):
        delta = delta / self.delta_level * self.w
        sigma = sigma / self.sigma_level * self.w
        V = self.smoothing_regularizer(delta, _delta, sigma, _sigma)
        D = 0
        pixel_group = self.D[point]
        for pixel in pixel_group:
            distance = ((pixel[0] - point[0]) ** 2 + (pixel[1] - point[1]) **
2) ** 0.5
            if self.trimap[pixel[0]][pixel[1]] == 0:
                distance = -distance
            alpha = self.distance_to_alpha(distance, sigma, delta)
            # print(alpha)
            tmp = self.data_term(alpha, point)
            D += tmp
            # print(tmp)
        if energy > V + D:
            # print("energy: ", energy)
            energy = V + D
            best_delta = delta
            best_sigma = sigma
```
4. Construct the alpha map with [0, 1] partition and multiply this map to the foreground image.

```
# Construct the alpha map
for point in self.C:
    delta, sigma = self.delta_sigma_dict[point]
    pixel_group = self.D[point]
    for pixel in pixel_group:
        distance = ((pixel[0] - point[0]) ** 2 + (pixel[1] - point[1]) ** 2)
** 0.5
        if self.trimap[pixel[0]][pixel[1]] == 0:
            distance = -distance
        alpha = self.distance_to_alpha(distance, sigma, delta)
        # print(alpha)
        alpha_map[pixel[0]][pixel[1]] = alpha
    distance = 0
    alpha = self.distance_to_alpha(distance, sigma, delta)
    alpha_map[point[0]][point[1]] = alpha
```
Finally, show the border matted image:

```
alpha_map = np.array(alpha_map)
border_matting_image = (alpha_map[:, :, np.newaxis] *
self.cropped_image).astype(np.uint8)
cv.imshow("Aftet Border Matting", border_matting_image)
```

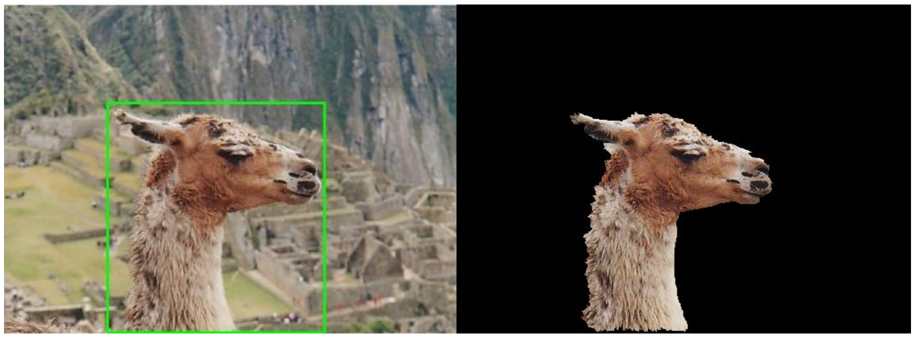
5 Summary

Realized: User interactive bounding box selection and finally output the GrabCut iterated foreground image and border matted image.

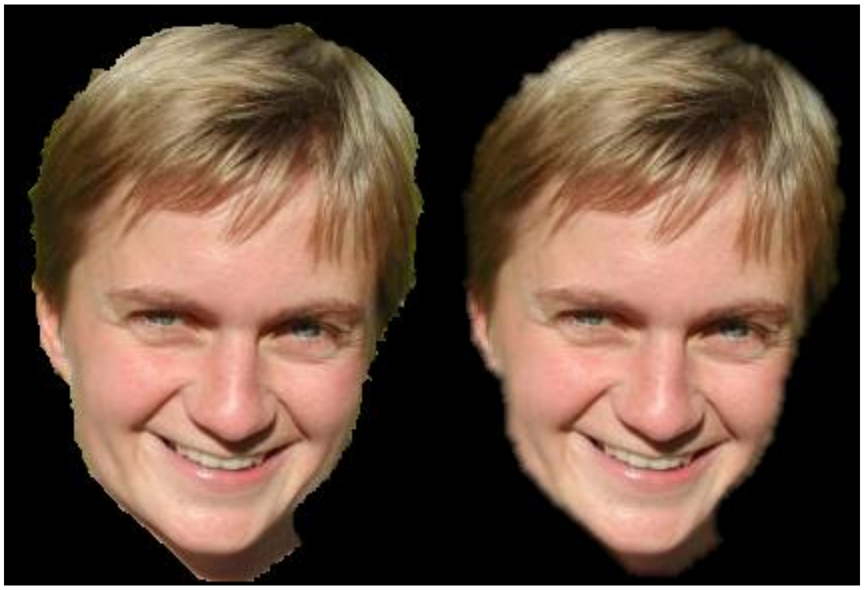
Unrealized: The iteration hasn't converged until the energy minimum but a fixed iteration, the foreground and background brush interactions haven't implemented, some functions in GrabCut and border matting still need to accomplish by hand crafted, the energy function costs too much power to calculate.

Appendices

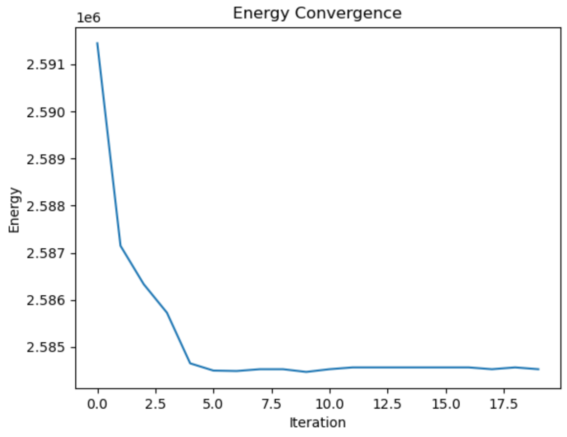
Repository: https://github.com/RexRusk/GrabCut/



GrabCut segmentation



Left: before border matting; Right: after border matting, it works but not works well



The energy function still needs to be completed

Ref

Rother, Carsten, Vladimir Kolmogorov, and Andrew Blake. "" GrabCut" interactive foreground extraction using iterated graph cuts." ACM transactions on graphics (TOG) 23.3 (2004): 309-314.

Boykov, Yuri Y., and M-P. Jolly. "Interactive graph cuts for optimal boundary & region segmentation of objects in ND images." Proceedings eighth IEEE international conference on computer vision. ICCV 2001. Vol. 1. IEEE, 2001.

Dempster, Arthur P., Nan M. Laird, and Donald B. Rubin. "Maximum likelihood from incomplete data via the EM algorithm." Journal of the royal statistical society: series B (methodological) 39.1 (1977): 1-22.

Kolmogorov, Vladimir, and Ramin Zabin. "What energy functions can be minimized via graph cuts?." IEEE transactions on pattern analysis and machine intelligence 26.2 (2004): 147-159.