An Implementation of the "Live-Wire" Image Segmentation Tool Using Java and OpenCV

Based on the papers "Interactive live-wire boundary extraction" and "Interactive Segmentation with Intelligent Scissors" by William A. Barrett and Eric N. Mortensen

James Gomez
CS 269
Final Project Report
Professor Terzopoulos

***Abstract*—Fully automated image segmentation for general imagery is still an unsolved problem. On the other hand, manual segmentation is tedious, time-consuming, and often inaccurate. In their paper "Interactive live-wire boundary extraction", Barrett and Mortensen introduced "live-wire", an interactive tool for fast, accurate, and reproducible image segmentation via mouse gestures. They refined their tool in 1998, this time referring to it as "intelligent scissors. This paper outlines an implementation of the live-wire tool using Java and the OpenCV computer vision library.**

I. INTRODUCTION

Image segmentation via manual boundary tracing is tedious, time-consuming, and often inaccurate. On the other hand, accurate, fully-automated general image segmentation is still not a fully-solved problem. As a result, it is desirable to have interactive tools available that enable users to extract segments of interest from images. These types of tools are especially desirable in the medical field, where extraction of specific objects from medical imagery is of high interest and importance. In 1996, Barrett and Mortensen introduced Live-wire, an interactive tool for fast and reproducible image segmentation [1]. Live-wire, also known as "intelligent scissors", frees users from the tedious job of extracting segments with manually drawn boundaries and improves both the speed and accuracy with which they are extracted.

This paper outlines a specific implementation of Barrett and Mortensen's Live-wire tool using the Java programming language and the OpenCV computer vision library. The remainder of this paper is organized as follows: in Section II, the previous work by Barrett and Mortensen is presented. In section III, the methodology and implementation details of this work are presented. Finally, in section IV, the paper is concluded.

II. PREVIOUS WORK

The following subsections briefly outline the methodology of Barrett and Mortensen's Live-wire tool.

A. Boundary Extraction

Barrett and Mortensen outline several steps that Live-wire follows in order to extract boundaries from images. First, a local cost map must be generated such that those pixels with strong edge features are associated with low costs and vice versa. Secondly, the local costs must be expanded from a user specified seed-point in order to generate least-cost paths from every pixel back to the seed-point. Next, real-time boundary detection and drawing is accomplished by traversing the least-cost path from the user's current cursor position back to the seed-point. Finally, the application detects closure of a boundary and extracts the pixels of that boundary.

i. Local Costs

Barrett and Mortensen describe the local cost function as the weighted sum of the gradient magnitude (f_G), gradient direction (f_D), and Laplacian zero-crossing (f_Z) features.

As described in [1], if we let $l(p, q)$ represent the local cost for the directed link from pixel p to pixel q, then the local cost function is

$$l(p, q) = \omega_G \cdot f_G(q) + \omega_Z \cdot f_Z(q) + \omega_D \cdot f_D(p, q)$$

where each ω is the corresponding feature weight. Their empirical default weight values were $\omega_G = 0.43$, $\omega_Z = 0.43$, $\omega_D = 0.14$. This gives us a cost map with high values at edge pixels. What we really need, however, is

low costs at edge pixels. Thus, the cost map is inverted and scaled appropriately.

In [2], the authors introduce three additional features for generating the local costs: Edge pixel value, "inside" pixel value, and "outside" pixel value. However, this implementation does not make use of these features.

i. Boundary Detection via Graph Search

The next step in the authors' method is to expand the local costs from a user-specified seed-point. This is done by performing a variation of Dijkstra's algorithm on a graph, where each node represents a pixel in the original image.

The node cost values are initialized with the local cost map from the previous section. The seed-point is then set to have a cost of 0. Starting at the seed-point, the costs are summed into its neighboring nodes, and each neighbor's parent pointer is set to point back to the expanding node if the new cumulative cost is less than its previous minimum cost or if the node does not yet have a parent pointer. The neighboring node with the least minimum cost is then expanded and the process is repeated, producing a wavefront of expansion, until all nodes have been expanded. After expansion, a minimum-cost path can be traversed from every pixel back to the original seed-point. For more details and illustrations on the expansion process, refer to [1] or [2].

Once the graph has been fully expanded, the Live-wire app is free to trace the boundary path back to the original seed point for whichever pixel within the image that the user's cursor is currently over. This allows for real-time manipulation and guidance of the boundary by the user.

The final step is to detect closure of a boundary. Although in [2] the authors briefly reference previous works that accomplish this task, they do not explicitly explain their approach. Thus, a rudimentary closed boundary detection algorithm is implemented here and described in section III.

B. Additional Features

To increase the robustness of Live-wire, Barrett and Mortensen introduced "path cooling" and "on-the-fly training". Though this implementation does not incorporate these features, they are described briefly here.

i. Path Cooling

Path cooling was introduced to free the user from the somewhat tedious job of manual seed-point placement. Path cooling automatically generates seed points for the user based on path coalescence over time [1]. This essentially freezes a previous portion of the boundary, which is no longer subject to change from then on. This allows a user to guide the free end of the boundary without having to manually generate the seed-points as he goes. For more details on path cooling, refer to [1] and [2].

ii. On-the-fly Training

Often, a user may be interested in extracting a boundary whose edge pixels have relatively weak edge features as compared to other nearby boundaries. In this case, the Live-wire boundary will tend to snap to the strong edge instead of the desired weaker edge. On-the-fly training was introduced to help overcome this problem.

On-the-fly training allows for dynamic adaptation of the underlying cost function based on previous sample boundary segments that have been considered "good" or "desirable" [1]. This is achieved by building a distribution of a variety of features from the most recent pixels of the most recent boundary segment and weighting similar features more heavily [1].

III. METHODOLOGY AND IMPLEMENTATION

In order to keep from re-inventing the wheel, it was desirable to use a software library or platform with common image processing and computer vision algorithms readily available. It was also desirable to use a language and environment that provides both good application performance (especially during graph expansion) and implementation speed and safety. Matlab provides an easy to use environment with high implementation speed and under-the-hood memory management, but does not offer the real-time performance required by this application. On the other hand, OpenCV, though powerful, has a standard C++ API interface. Though C++ has the quality of compiling to high performance native code, it provides a tedious implementation experience relative to other high level language alternatives, as memory must be explicitly managed by the programmer. Thus, this live-wire implementation is based in Java and interfaces the

OpenCV libraries via the JavaCV wrapper classes and APIs. Java provides the real-time performance needed for this application while offering a safer and higher speed implementation environment than C++.

The process of extracting a closed boundary involves several steps, which were described previously in section II. The following subsections detail the implementation of those steps for this work.

A. Feature Extraction and Cost Map Generation

i. Gradient Features

Prior to extraction of gradient features, a Gaussian blur is performed on the image in order to reduce high frequency noise. OpenCV provides several methods for gradient feature extraction. For this implementation, the Sobel method was used to extract the individual x and y gradient components from the image.

In order to extract the gradient magnitude from the gradient components, we first take the absolute value of the x and y gradient components. Once these are obtained, we calculate a weighted sum of the absolute value of the components to obtain an approximation of the gradient magnitude.

Since the Sobel operator returns image components with negative values, we must shift and scale the values of the gradient components' pixels to fit a single channel, 8-bit grayscale image matrix in order to extract the gradient direction feature. Once converted, we can obtain the gradient direction of the image per pixel by taking the arctangent of the quotient of the corresponding pixel in the y gradient component and the corresponding pixel in the x gradient component, i.e. $\arctan(\frac{gradY}{gradX})$. Figures 1 through 5 on the following page provide visualizations of the extracted gradient features.

ii. Canny Edges

Since the Laplacian zero-crossing feature described by Barrett and Mortensen is a binary edge feature, it is reasonable to use any similar gradient-based method that also produces binary edges. OpenCV provides several methods for extracting edges from an image, including Laplacian based methods. This implementation, however, makes use of OpenCV's implementation of the Canny edge detector. The image is smoothed via a Gaussian blurring filter before extraction of edges to again reduce high frequency noise. Once smoothed, the Canny edge detector is used to extract a binary image of the edges from the original image. Figure 6 shows the binary image of edges extracted by the Canny edge detector.

iii. Cost map

Once the Gradient and edge features have been extracted, we can generate an initial local cost map by taking a weighted sum of the features and inverting the result. This implementation makes use of the author's empirical weights from [1] since experimental runs show that these values generally give good results. Figure 7 shows the resulting cost map.

B. Graph Expansion

After the local cost map has been generated from the extracted image features, we can use the cost map as a starting point for the graph expansion process.

iv. Seed-point snapping

To start the expansion process, a user clicks on a point in the image to specify a starting seed-point. As mentioned in [1] and [2], specifying a seed-point that actually lies on an edge boundary can often be tedious. To aid the user in specifying an accurate seed-point, seed-point snapping was implemented.

Seed-point snapping works by taking the users click point and searching an area of pixels (7 pixel radius in this implementation) around that point. Since those pixels with the lowest local costs reside on edges, the pixel with the lowest initial local cost is used as the actual seed-point to start the expansion process.

v. Expansion

Once a good seed-point has been extracted, we can begin the expansion process. To simplify implementation, data structures from the Java Collections Framework were used. A PriorityQueue is created to keep track of the expanding wavefront of pixel-nodes, while a HashSet was used to keep track of the already expanded nodes. However, since PriorityQueue only offers $O(n)$ performance for its *contains()* method, an additional HashSet was created containing a mirror of the nodes in the wavefront PriorityQueue. Since HashSets

Figure 1: An original image of a banana

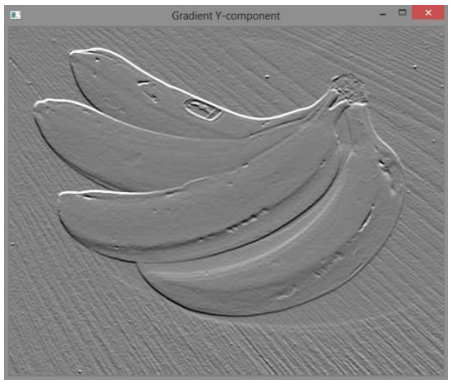

Figure 2: The y-component of the gradient of the image in figure 1

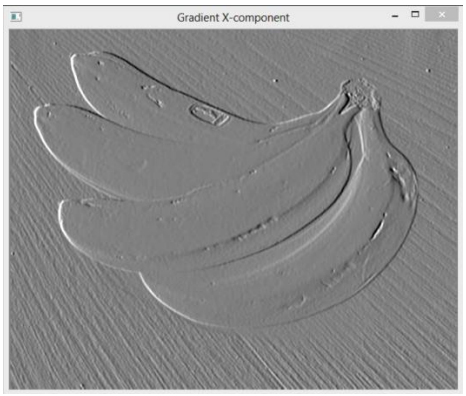

Figure 3: The x-component of the gradient of the image in figure 1

Figure 4: The gradient magnitude of the image in figure 1

Figure 5: A visualization of the gradient direction of the image in figure 1

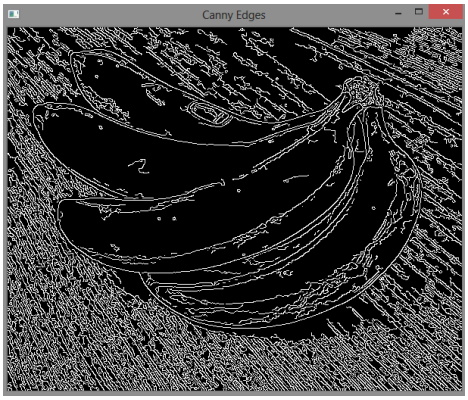

Figure 6: The Canny edges extracted from the image in figure 1

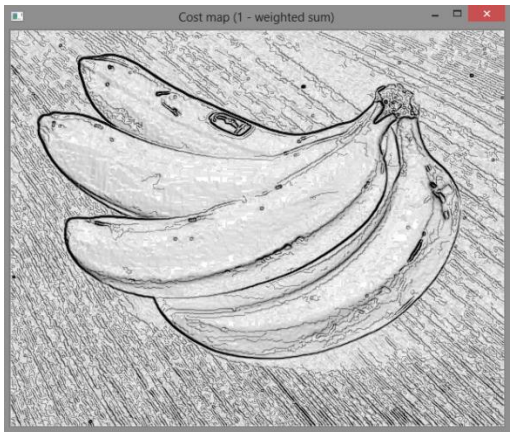

Figure 7: A visualization of the inverted weighted sum of the gradient magnitude, gradient direction, and Canny edge features.

have constant time lookup, this provided dramatic speedup of the expansion process at the cost of additional memory. The result is expansion of small and medium sized images on the order of milliseconds, and expansion of larger images in less than 2 seconds.

C. Live-wire Boundary Tracing

Once the image graph has been expanded from the seed-point, real-time boundary tracing can take place. This is accomplished by taking the user's cursor position for any given frame and following the path of parent pointers back to the seed-point, drawing line segments over the image along the way.

Figures 8 and 9 show some examples of boundary tracing. In Figure 8, a live portion of the boundary, shown in red, is being traced from the user's cursor (the lower portion of the boundary) back to an initial seed-point (the top portion of the boundary). In Figure 9, the user has cooled a previous portion of the boundary, shown in cyan, by generating a new seed-point manually, and the live portion of the boundary is being traced back to that new seed-point, shown in red. Whenever a portion of the boundary is cooled, the pixels within the cooled portion of the boundary are added to a final list of boundary points to be used for segment extraction later.

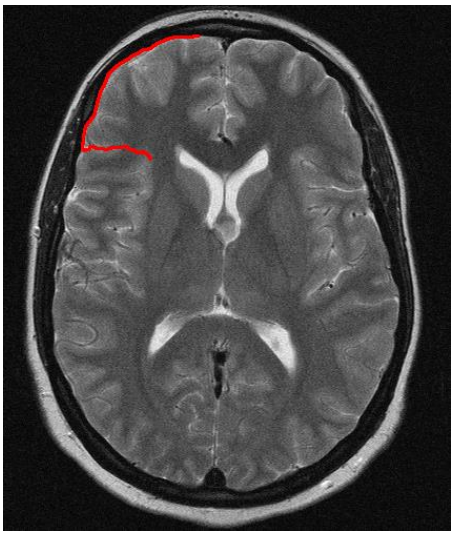

Figure 8: Live-wire initial boundary

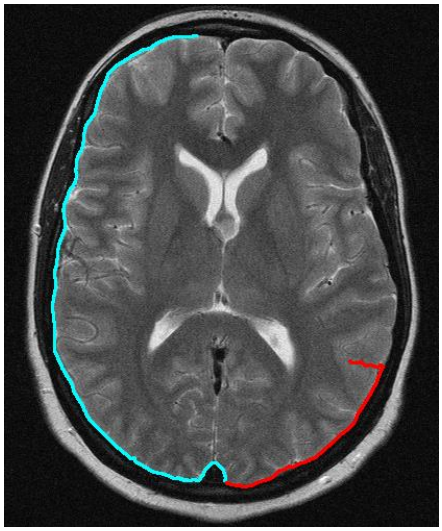

Figure 9: Live-wire boundary with cooled boundary portion

D. Closed Boundary Detection

In order to extract a segment from the image, it is necessary to detect when a boundary has been closed. Whenever a user generates a seed-point, the portion of the boundary that was live becomes cooled. During cooling, for every point along the currently live boundary, we test if that point overlaps with the original starting seed-point. If any of the points along the current boundary correspond with the original seed-point, then the boundary has overlapped itself and has closed off a segment.

For most cases, this simple overlap detection works well. However, in some cases, it is possible that the live portion of the boundary overlaps another portion of the boundary that does not include the original seed-point. In these cases, the app will not detect a closed boundary and will continue to draw a live boundary. In most cases where this occurs, it is easiest to clear the current boundary and start over with a new boundary.

When a closed boundary is detected, all pixels along the live portion of the wire that crossed beyond the original seed-point are removed from the list of boundary points, leaving a complete, non-redundant set of pixel points as the closed boundary.

Figure 10 below shows a closed boundary after overlap has been detected.

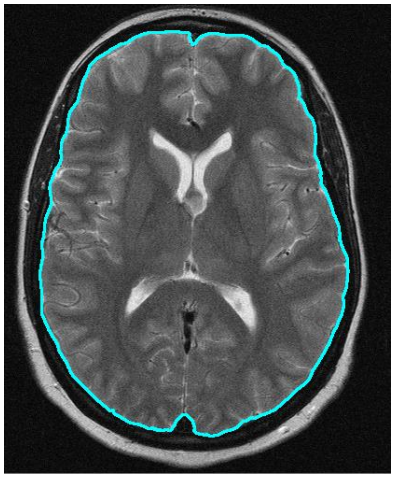

Figure 10: Closed boundary after overlap detection

E. Segment Extraction

The final step is to use the list of boundary points from step D to extract the image segment. OpenCV provides a method, *cvPointPolgonTest()*, for detecting whether a point resides within a closed contour or not. Using this method and our list of boundary points as our contour definition, for every pixel *p* within the image, we test if *p* lies inside or along the boundary of the contour. If it does, we write the value of that pixel to the corresponding pixel in a separate image buffer. Otherwise, we write a value of 0 to the corresponding pixel of the image buffer. Once complete, the image buffer will contain only those pixels that were bound by the live-wire boundary, leaving the rest of the pixels black. Left double-clicking over the image allows the user to save the extracted boundary and segment. Figures 11 and 12 show images of the extracted boundary and segment, respectively.

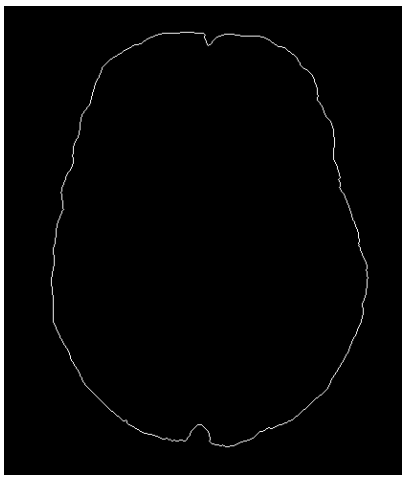

Figure 11: Extracted boundary

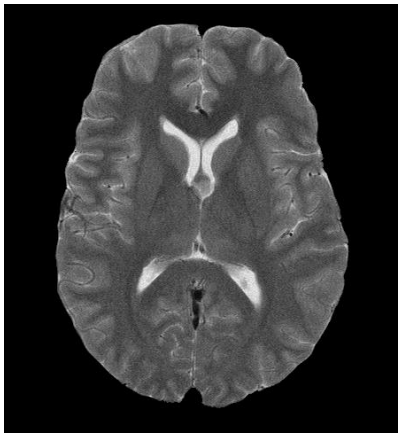

Figure 12: Extracted segment

IV. CONCLUSION

This paper has outlined an implementation of the live-wire image segmentation tool by Barrett and Mortensen using Java and OpenCV. All basic functionality has been implemented, including graph expansion, real-time boundary tracing, closed boundary detection, and segment and boundary extraction. Automatic path cooling and on-the-fly learning, however, remains unimplemented.

Overall performance of the application is good, taking on the order of milliseconds to perform graph expansion on small to medium-sized images, and less than 2 seconds on larger images that fit within the dimensions of a standard 1920x1080 monitor. Boundary tracing thereafter performs at real-time speed.

The algorithm for closed boundary detection in this implementation is limited, and could be expanded upon by enabling the app to detect the overlap of the live boundary with any portion of the cooled boundary. This has the potential to improve robustness drastically.

REFERENCES

[1] W. A. Barrett and E. N. Mortensen, "Interactive Segmentation with Intelligent Scissors," *Graphical Models and Image Processing*, pp. 349-384, 1998.

[2] W. A. Barrett and E. N. Mortensen, "Interactive live-wire boundary extraction," *Medical Image Analysis*, vol. 1, pp. 331-341, 1996/7.