Interactive Segmentation with Intelligent Scissors

Eric N. Mortensen

William A. Barrett

Brigham Young University

Abstract

We present a new, interactive tool called *Intelligent Scissors* which we use for image segmentation. Fully automated segmentation is an unsolved problem, while manual tracing is inaccurate and laboriously unacceptable. However, Intelligent Scissors allow objects within digital images to be extracted quickly and accurately using simple gesture motions with a mouse. When the gestured mouse position comes in proximity to an object edge, a *live-wire boundary* "snaps" to, and wraps around the object of interest.

Live-wire boundary detection formulates boundary detection as an optimal path search in a weighted graph. Optimal graph searching provides mathematically piece-wise optimal boundaries while greatly reducing sensitivity to local noise or other intervening structures. Robustness is further enhanced with *on-the-fly training* which causes the boundary to adhere to the specific type of edge currently being followed, rather than simply the strongest edge in the neighborhood. *Boundary cooling* automatically freezes unchanging segments and automates input of additional seed points. Cooling also allows the user to be much more free with the gesture path, thereby increasing the efficiency and finesse with which boundaries can be extracted.

1. Introduction

Fully automatic general image segmentation is an unsolved problem due to the wide variety of image sources, content, and complexity and hence, has given way to a variety of semiautomated approaches, initialization schemes, etc. In many cases, manual segmentation (i.e., tracing the object boundary) is still widely used when an image component must be segmented from a complex background. For this reason intelligent segmentation tools which exploit high level visual expertise but require minimal user interaction become appealing.

This paper details an interactive, digital image segmentation tool called "Intelligent Scissors" which allows rapid object extraction from arbitrarily complex backgrounds. Intelligent Scissors formulates boundary finding as an unconstrained graph search [11] in which the boundary is represented as an optimal path within the graph. The main advantage of this technique, which differentiates it from previous optimal boundary based segmentation techniques, is the interactive "live-wire" tool, developed in our lab in January of 1992. The live-wire tool allows the user to interactively select an optimal boundary segment by immediately displaying the minimum cost path from the current cursor position to a previously specified "seed" point in the image. Thus, it is the method and style of interaction which fundamentally distinguishes our Intelligent Scissors technique from previous work in optimal boundary detection.

Some boundary based segmentation techniques compute a single optimal boundary based on some initial template or contour. Rather than decide on a single optimal boundary, our live-wire technique computes, at interactive speed, an optimal path from a selected seed point to *every* other point in the image and lets the user choose, interactively, based on the current cursor position, the path which visually corresponds best to a segment of the desired object boundary.

To minimize user interaction required in manual seed point selection, seed points are generated automatically along a current active boundary segment via boundary "cooling". Boundary cooling occurs when a section of the current interactive portion of the boundary has not changed recently and consequently "freezes", depositing a new seed point, while reinitiating the optimal path expansion.

Training on boundary characteristics (gradient magnitude, image intensity, etc.) is also added. To allow the algorithm to adapt to different types of edges, training is implemented dynamically rather than having a separate training phase. This allows the live-wire tool to adjust to changing boundary characteristics *during* segmentation.

Figure 1 demonstrates how live-wire segmentation is used to extract a complex object boundary from a nontrivial background. Figures 1(a-c) show selected frames from the first couple of seconds of the interactive segmentation process while Figures 1(d-f) are frames near the end of the process. Fig. 1(g) is the completed boundary. The entire boundary was defined in approximately 45 seconds. Fig. 1 also provides a feel for the interactive style of the live-wire segmentation process. Notice in frames (a) through (f) that a seed point (shown as a red dot), is "anchored" to the boundary of the object and that another point (indicated by the green cross hairs) is free to move around the image. Since a globally optimal path is computed at interactive speeds from a seed point to every other point in an image, that path is displayed as the "free" point moves, thereby allowing the user to interactively select an optimal path that corresponds to a portion of the desired boundary. This interactive selection is the essence of the live-wire and is at the heart of Intelligent Scissors.

The remainder of this paper details previous optimal boundary based techniques, the Intelligent Scissors tool, and presents both qualitative (visual) and quantitative (timing, accuracy, and reproducibility) results obtained with the Intelligent Scissors segmentation tool.

(a) (b) (c)

(d) (e) (f)

(g)

Figure 1: *Selected frames from an example live-wire segmentation where both the object boundary extracted and the background are complex. The red dots are seed points, the green crosshair is the free point, the blue contour segments correspond to portions of the "set" boundary, and the yellow contour segment is the live-wire boundary segment. (a-c) Selected frames from the first couple of seconds of the interactive segmentation process. (d-f) Selected frames from the last couple of seconds of the process. (g) The final object boundary contains 2348 pixels. (Image size: 640×420)*

2. Previous Work in Optimal Boundary Detection

Among the global boundary based techniques, graph searching and dynamic programming are popular techniques used to find the globally "optimal" boundaries based on some local cost criteria [2,7,8,20-23]. They formulate the boundary finding problem as a directed graph or cost functional to which an optimal solution is sought.

Montanari [21] was perhaps the first to apply a global optimization algorithm to boundary detection in images. He proposes "a technique for recognizing systems of lines" based on dynamic programming to minimize a heuristic "figure of merit" or cost function and develops a figure of merit for low curvature lines based on image intensity, path curvature and path length. The algorithm detects a line (with local variations) in an artificial image even when the line is hardly visible due to noise.

Ballard and Sklansky [2] extend Montanari's algorithm by using gradient magnitude, gradient direction and a closure measure in their evaluation function. They use dynamic programming with directed searching to detect circular tumors in chest radiographs.

Chien and Fu [8] argue that Ballard and Sklansky's decision function is "too specifically designed for one type of application" and develop a more general "criterion" function which has both local (e.g., gradient) and global (e.g., curvature) components. They minimize the criterion function using a modified decision tree search and apply their technique to determine cardiac boundaries in chest x-rays.

Martelli [20] shows that any optimization problem using dynamic programming can be formulated as a shortest or minimum cost path graph search. He applies Nilsson's [27] A* heuristic graph search algorithm to the boundary detection problem where the heuristic is used to prune the search and thereby reduce computation. His technique successfully identifies multiple touching objects and occluded boundaries in artificial images with Gaussian noise.

Cappalletti and Rosenfeld [7] also use Nilsson's A* algorithm with searching constraints to extract closed, two-dimensional boundaries in each slice of a three-dimensional volume. The local cost is a function of the three-dimensional gradient with an additional distance cost from any

contour in a neighboring slice. Graph searching is applied iteratively, both in 2-D and 3-D, to improve the boundary in each iteration.

Udupa [28] formulates optimal two-dimensional boundary finding as a graph search using dynamic programming. Like Martelli [20], he formulates the image grid as a directed graph where pixel corners are nodes and the cracks between pixels are directed arcs. Unlike previous approaches, his formulation does not impose any sampling constraints on the boundary shape or searching constraints within the graph search, allowing paths of arbitrary complexity to be extracted. The algorithm computes cumulative graph node costs in a step-wise dynamic programming fashion until an optimal path is computed from a seed node to every other node within the image (or some specified region of interest).

Based on Udupa's formulation, and in collaboration with him, Morse et al. [22,23] present a boundary finding algorithm which computes a piece-wise optimal boundary given multiple input control points. Rather than specify constraints and heuristics for a specific problem, this method utilizes a probabilistic "likelihood" function. Manual training provides specific feature distributions used to compute Bayesian probabilities. Since the algorithm iterates through the 2-D likelihood matrix (generated from the input image), the resulting complexity is $O(n^3)$ where n is the image width and height. It is important to emphasize that, unlike the live-wire technique presented in this paper, the approach in [22,23] is strictly an iterative, non-interactive method for boundary finding, where a series of user-selected control points are fed into a dynamic programming procedure, requiring a few 10's of seconds to compute the boundary.

Snakes, active contours, and thin plate models are another global boundary based segmentation techniques that have received a great deal of attention [1,9,10,14,17,18,30]. Active contours are initialized manually with a rough approximation to a boundary of interest and then allowed to iterate over the contour to determine the boundary that minimizes an energy functional.

Kass, Witkin, and Terzopoulos [17,18] introduced a global minimum energy contour called "snakes" or active contours. Given an initial approximation to a desired contour, a snake locates the closest minimum energy contour by iteratively minimizing an energy functional which

combines internal forces to keep the active contour smooth, external forces to attract the snake to image features, and constraint forces which help define the overall shape of the contour. The constraint forces are applied by interactively attaching “springs” to points on the contour which pull on the contour or by placing “volcanoes” (high energy peaks) which repel the contour. Points of high curvature can be specified since curvature weights are parametric. The energy minimization is performed via variational calculus.

Amini, Weymouth, and Jain [1] show that variational calculus used by Kass et al. may be subject to relative (local) minima, that it cannot enforce hard (non-differentiable) constraints, that it may be numerically unstable for discrete, noisy data, and finally that it may oscillate. They present an active contour algorithm with a similar energy functional but use dynamic programming to minimize the functional rather than variational calculus.

Williams and Shah [30] claim that the dynamic programming technique of Amini et. al. is too time and memory expensive, being $O(nm^3)$ for both. They propose a locally optimal technique that minimizes the energy functional local to each contour point. This results in faster iterations but may require more iterations to converge. Their results compared well to the variational calculus technique when applied to contrived images that contained strong, well defined edges and simple object shapes.

Daneels et al. [10] compare the active contour methods presented by Kass et al., Amini et al., and Williams and Shah in terms of iteration speed, number of iterations, and quality of results. They then propose a two-stage technique that uses the greedy algorithm presented by Williams and Shah for quick initial convergence followed by a few iterations of the slower dynamic programming technique (optimized by alternating search neighborhoods and dropping “stable” points) for improved quality.

Geiger et al. [14] apply dynamic programming to detect deformable contours. They use a noniterative technique that searches for the optimal contour within a large neighborhood around the initial contour. They utilize a multi-scale technique to achieve greater processing efficiency while sacrificing guaranteed optimality. Like Kass et. al. [17,18], they apply deformable contours

to the detection of object boundaries, tracking object boundaries over time, and matching object boundaries in stereo pairs.

Cohen and Kimmel [9] utilize a shortest path approach (similar to [11]) to detect the global minimum of an active contour's energy between two points. Like [14], their approach does not iterate over the contour, but rather they find the single, globally optimal solution for all paths connecting two points, thereby reducing initialization time. They compute a "surface of minimal action" from one point $\boldsymbol{p}_0$ to every other point in the image then employ gradient descent to determine an optimal path from the another fixed point, $\boldsymbol{p}_1$. They can also detect closed boundaries in an image given a single boundary point by determining minimal saddle points in the surface.

The methods discussed thus far follow a pattern of user input--whether through defining a figure of merit, a decision function, a 2-D template, or an initial active contour, etc.--to initialize the algorithm, followed by contour selection based on the input, for the graph searching techniques, or contour refinement of the input, for active contour techniques. If the resulting contour is not satisfactory, this may in turn be followed by one or more iterations of user input (to adjust parameters, change the figure of merit, input a new initial active contour, locally modify an existing contour or energy landscape, etc.) and reapplication of the algorithm.

This cycle exists because the previous algorithms often compute a single contour based on the user input. An alternative approach would be to compute multiple candidate contours (or contour segments) and then let the user select the desired contour interactively from the candidate set. With the possible exception of [9], such an approach would be problematic for most of the active contour models since they require an initial contour (or a piece of one [10]) on which to iterate. On the other hand, the graph searching methods have the inherent capability to compute an optimal path to multiple destination nodes. In terms of image space, such algorithms can compute an optimal path from a start point to many, if not all, pixels specified within a sampling window (as defined by any geometric heuristics or 2-D templates). Unfortunately, previous graph searching techniques typically limit the utility of the optimal computation by requiring goal nodes to be specified *a priori*.

The interactive optimal path selection algorithm, or live-wire technique, was developed as a general image segmentation tool which takes advantage of the multiple optimal paths generated by graph searching techniques. The live-wire technique was developed to overcome the limitations of [22,23]. Although [22,23] use dynamic programming to compute unrestricted optimal paths from every grid point in the image to every other grid point, it still suffers from the same iterative, non-interactive style as previous graph searching boundary finding methods in that the user inputs a series of control points which are then connected with piece-wise optimal segments into a single contour. There is no immediate feedback to indicate where, or how far apart to place the seed points on the boundary. Consequently, multiple iterations, requiring input of multiple control points is typical with this technique.

The live-wire technique eliminates the guess work of previous “batch mode” seed point placement methods by providing immediate, interactive feedback of the optimal boundary segment as the user places each point. Both the concept of interactive optimal path selection and the term “live-wire” had their origin at Brigham Young Univ. in Jan. 1992. A working prototype of the live-wire tool was demonstrated to Udupa in Feb. 1992 [4]. Each group pursued separate implementations independent of each other which were subsequently presented almost concurrently in conference proceedings [24,29]. Continued independent development of the live-wire algorithm appears in various conference proceedings [5,12,25] as well as a Master’s thesis [26].

While Intelligent Scissors and the live-wire algorithm has been published previously [5,6,24,25], it has been in limited form due to page limitations. Thus, the purpose of this paper is to present the full details of Intelligent Scissors: the local cost functional (with on-the-fly training), the efficient implementation of Dijkstra’s optimal graph search [11], and especially the interactive live-wire optimal path selection tool (with cursor snap and path cooling). Further, this paper presents quantitative timing, accuracy, and reproducibility results and compares them to manual tracing.

3. Intelligent Scissors

The underlying mechanism for Intelligent Scissors is the "live-wire" path selection tool. The live-wire tool allows the user to interactively select the desired optimal path from the entire collection of optimal paths (one for each pixel in the image) generated from a specified seed point. The optimal path from each pixel is determined at interactive speeds by computing an optimal spanning tree of the image using an efficient implementation of Dijkstra's graph searching algorithm. The basic idea is to formulate the image as a weighted graph where pixels represent nodes with directed, weighted edges connecting each pixel with its 8 adjacent neighbors.

3.1 Local Costs

If $\boldsymbol{p}$ and $\boldsymbol{q}$ are two neighboring pixels in the image then $l(\boldsymbol{p}, \boldsymbol{q})$ represents the local cost on the directed link (or edge) from $\boldsymbol{p}$ to $\boldsymbol{q}$. The local cost function is a weighted sum of component cost functions on each of the following image features:

Image Feature	Formulation
Laplacian Zero-Crossing	f_Z
Gradient Magnitude	f_G
Gradient Direction	f_D
Edge Pixel Value	f_P
"Inside" Pixel Value	f_I
"Outside" Pixel Value	f_O

Combining these feature components into a local cost function gives

$$l(\boldsymbol{p}, \boldsymbol{q}) = \omega_Z \cdot f_Z(\boldsymbol{q}) + \omega_G \cdot f_G(\boldsymbol{q}) + \omega_D \cdot f_D(\boldsymbol{p}, \boldsymbol{q}) + \omega_P \cdot f_P(\boldsymbol{q}) + \omega_I \cdot f_I(\boldsymbol{q}) + \omega_O \cdot f_O(\boldsymbol{q}) \qquad (1)$$

where each ω is the weight of the corresponding feature function. Empirically (and by default), weights of $\omega_Z = 0.3$, $\omega_G = 0.3$, $\omega_D = 0.1$, $\omega_P = 0.1$, $\omega_I = 0.1$, and $\omega_O = 0.1$ seem to work well in a wide range of images. However, these weights can be easily adjusted.

The Laplacian zero-crossing, f_Z, and the two gradient features, f_G and f_D, have static cost functions. Static costs can be computed without any *a priori* information about image content. The gradient magnitude, f_G, and the three pixel value components, f_P, f_I, and f_O, ("inside" and "outside" features are introduced in [12,29]) have dynamic cost functions. (Note that f_G is the only cost feature that has both static and dynamic components.) Dynamic costs can be computed only after training [3] (discussed in Section 3.1.6). Since a meaningful static cost function for f_P,

f_I, and f_O could not be formulated, they have meaning only after training. As a result, if training is turned off or if no training data is available, the weights for f_P, f_I, and f_O are zero.

The Laplacian zero-crossing, f_Z, and the gradient magnitude, f_G, are edge operators employing image convolution with multi-scale kernels. This allows the cost functions for these features to adapt to a variety of image types by automatically selecting, on a pixel by pixel basis, the kernel width that best matches the line-spread function of the imaging hardware used to obtain the current image [13,16].

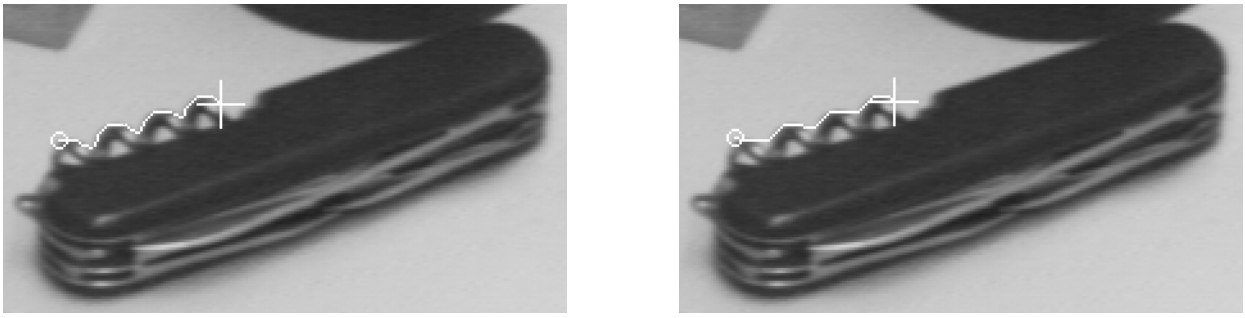
3.1.1 Laplacian Zero-Crossing (f_Z)

The primary purpose of the multi-scale Laplacian zero-crossing component, f_Z, is for edge localization [13,19]. As mentioned, multiple kernel widths are used, each corresponding to a different standard deviation for the 2-D Gaussian distribution. The kernels are normalized such that the sum of their positive elements (or weights) are equal. This is done so that comparisons can be made between the results of convolutions with different kernel sizes. The standard deviations used to compute Laplacian kernels vary from 1/3 of a pixel (producing a 5x5 kernel) to 2 pixels (giving a 15x15 kernel) in increments of 1/3 of a pixel. The kernels are large enough to include all kernel elements which are nonzero when represented as 16 bit fixed-point values. Multiple kernel sizes are used because smaller kernels are more sensitive to fine detail while larger kernels suppress noise. By default, kernel sizes of 5x5 and 9x9 are used and seem to work well in a variety of images. However, for low contrast, low SNR images, larger kernel sizes can be easily used.

The Laplacian zero-crossing is used to create a binary local cost feature. If a pixel is on a zero-crossing then the component cost for all links to that pixel is low; otherwise it is high. That is, if $I_L(\boldsymbol{q})$ is the Laplacian of the original image, I, at a point or pixel $\boldsymbol{q}$, then

$$f_Z(\boldsymbol{q}) = \begin{cases} 0; & \text{if } I_L(\boldsymbol{q}) = 0 \\ 1; & \text{if } I_L(\boldsymbol{q}) \neq 0 \end{cases} \tag{2}$$

However, a discrete Laplacian image produces very few, if any, actual zero valued pixels. Rather, a zero-crossing is represented by two neighboring pixels with opposite sign. Of the two pixels, the one that is closest to zero is chosen to represent the zero-crossing. Thus, f_Z is 0 for Laplacian image pixels that are either zero or closer to zero than any neighbor with an opposite sign; other-



(a) (b)

Figure 2: *(a) Optimal path with and (b) without the binary Laplacian zero-crossing local cost feature.*

wise, f_Z is 1. The four horizontal/vertical neighbors of a pixel constitute the neighborhood used to determine the zero-crossing. This creates a single pixel wide cost "canyon" and results in boundaries "snapping" to and localizing object edges.

Figure 2 demonstrates the difference between globally optimal boundaries defined with and without the zero-crossing feature. Notice how the optimal path defined with the zero-crossing cost canyon follows the corkscrew more tightly than does the path without the binary feature.

Since multiple kernels can be used in the formulation of f_Z, then each binary cost feature resulting from a given kernel width (or standard deviation) has a weight which contributes to the component feature cost. That is, the zero-crossing cost feature is the weighted sum of the binary zero-crossing maps computed for each kernel size used where the sum of the kernel weights is unity. (Default values are 0.45 for the 5x5 kernel and 0.55 for the 9x9 kernel). Therefore, a given pixel's zero-crossing feature cost, f_Z, is zero if and only if the Laplacian from each kernel gives a zero-crossing at that pixel and it is 1 if and only if all Laplacian outputs do not have a zero-crossing at that pixel, otherwise $0 < f_Z < 1$.

3.1.2 Multi-Scale Gradient Magnitude (f_G)

Since the Laplacian zero-crossing creates a binary feature, f_Z does not distinguish between a "strong" or high gradient edge and a "weak" or low gradient edge. Gradient magnitude, however, is directly proportional to the image gradient. The gradient magnitude is computed by approximating the partial derivatives of the image in x and y using derivative of Gaussian kernels of various scales. This gives the horizontal, I_x, and the vertical, I_y, partial gradient magnitudes of the image. An image's gradient magnitude G can then be approximated by $G = \sqrt{I_x^2 + I_y^2}$. How-

ever, the static gradient magnitude cost feature needs to be low for strong edges (high gradients) and high for weak edges (low gradients). Thus, the static cost feature is computed by subtracting the gradient magnitude image from its own maximum and then dividing the result by the maximum gradient (to scale the maximum cost to 1 prior to multiplying by the feature weight ω_G). The resulting static feature cost function is

$$f_G = \frac{\max(G') - G'}{\max(G')} = 1 - \frac{G'}{\max(G')} \tag{3}$$

where $G' = G - \min(G)$ for G computed above, giving an inverse linear ramp function.

As with the Laplacian zero-crossing, multiple kernel sizes are used to compute the gradient magnitude feature cost. Also, each kernel is normalized such that the sum of positive kernel values is equal for all kernel widths. This is done for the same reason as for the Laplacian kernels: so direct comparisons can be made between the results obtained from different kernel sizes.

Unlike the results of the multiple Laplacian kernels, the multiple gradient magnitude kernel results are not simply combined in a weighted linear fashion. Instead, the result for the kernel that "best" approximates the natural spatial scale of each particular edge, on a pixel by pixel basis, is used. Best match is estimated in one of two ways. First, the kernel size giving the largest gradient magnitude at a pixel is the kernel size used at that pixel. Or second, the Laplacian kernel producing the steepest slope at the zero-crossing corresponds to the best gradient magnitude kernel size for that point. By default, the second technique, based on the Laplacian kernel, is used to determine the best kernel size, but the first method can be specified (for low contrast, low SNR images where the zero-crossing information is noisy and unreliable).

Figure 3 shows the two gradient magnitude images obtained from Fig. 13(a) using both techniques for determining the best kernel size. Fig. 3(a) was computed by convolving every pixel with the gradient magnitude kernels for each kernel size and keeping the result that produced the largest magnitude. Fig. 3(b) uses the maximum Laplacian zero-crossing slope to determine which size of gradient magnitude kernel to apply at each pixel. Since the second technique only produces output for zero-crossing pixels, those pixels that do not correspond to a zero-crossing use, by default, the smallest (3x3) gradient magnitude kernel to provide a complete gradient map.

(a) (b)

Figure 3: *(a) Gradient image of Fig. 4.2(a) obtained by convolving each pixel with the gradient magnitude kernel that produced the largest magnitude. (b) Gradient image obtained by convolving with the gradient magnitude kernel size that produced the largest Laplacian zero-crossing slope.*

3.1.3 Gradient Direction (f_D)

The gradient direction or orientation adds a smoothness constraint to the boundary by associating a relatively high cost for sharp changes in boundary direction. The gradient direction is simply the direction of the unit vector defined by I_x and I_y. Therefore, letting $\boldsymbol{D}(\boldsymbol{p})$ be a unit vector of the gradient direction at a point $\boldsymbol{p}$ and defining $\boldsymbol{D}'(\boldsymbol{p})$ as the unit vector perpendicular (rotated 90° clockwise) to $\boldsymbol{D}(\boldsymbol{p})$ (i.e., for $\boldsymbol{D}(\boldsymbol{p}) = [I_x(\boldsymbol{p}), I_y(\boldsymbol{p})]$, $\boldsymbol{D}'(\boldsymbol{p}) = [I_y(\boldsymbol{p}), -I_x(\boldsymbol{p})]$), then the formulation of the gradient direction feature cost is

$$f_D(\boldsymbol{p}, \boldsymbol{q}) = \frac{2}{3\pi}\{\operatorname{acos}[d_{\boldsymbol{p}}(\boldsymbol{p}, \boldsymbol{q})] + \operatorname{acos}[d_{\boldsymbol{q}}(\boldsymbol{p}, \boldsymbol{q})]\} \tag{4}$$

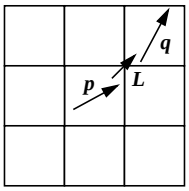
where

$$\begin{aligned} d_{\boldsymbol{p}}(\boldsymbol{p}, \boldsymbol{q}) &= \boldsymbol{D}'(\boldsymbol{p}) \cdot \boldsymbol{L}(\boldsymbol{p}, \boldsymbol{q}) \\ d_{\boldsymbol{q}}(\boldsymbol{p}, \boldsymbol{q}) &= \boldsymbol{L}(\boldsymbol{p}, \boldsymbol{q}) \cdot \boldsymbol{D}'(\boldsymbol{q}) \end{aligned} \tag{5}$$

are vector dot products and

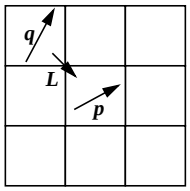
$$\boldsymbol{L}(\boldsymbol{p}, \boldsymbol{q}) = \frac{1}{\|\boldsymbol{p} - \boldsymbol{q}\|}\begin{cases} \boldsymbol{q} - \boldsymbol{p}; & \text{if } \boldsymbol{D}'(\boldsymbol{p}) \cdot (\boldsymbol{q} - \boldsymbol{p}) \geq 0 \\ \boldsymbol{p} - \boldsymbol{q}; & \text{if } \boldsymbol{D}'(\boldsymbol{p}) \cdot (\boldsymbol{q} - \boldsymbol{p}) < 0 \end{cases} \tag{6}$$

is the normalized bidirectional link or unit edge vector between pixels $\boldsymbol{p}$ and $\boldsymbol{q}$ and simply computes the direction of the link between $\boldsymbol{p}$ and $\boldsymbol{q}$ such that the difference between $\boldsymbol{p}$ and the direction of the link is minimized. Links are either horizontal, vertical, or diagonal (relative to the position



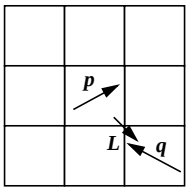
(a)

$\boldsymbol{D}'(\boldsymbol{p}) = [0.870, 0.492]$
$\boldsymbol{D}'(\boldsymbol{q}) = [0.473, 0.881]$
$\boldsymbol{L}(\boldsymbol{p},\boldsymbol{q}) = [0.707, 0.707]$
$d_{\boldsymbol{p}}(\boldsymbol{p},\boldsymbol{q}) = \boldsymbol{D}'(\boldsymbol{p}) \cdot \boldsymbol{L}(\boldsymbol{p},\boldsymbol{q}) = 0.964$
$d_{\boldsymbol{q}}(\boldsymbol{p},\boldsymbol{q}) = \boldsymbol{L}(\boldsymbol{p},\boldsymbol{q}) \cdot \boldsymbol{D}'(\boldsymbol{q}) = 0.957$
$f_D(\boldsymbol{p},\boldsymbol{q}) = \frac{2}{3\pi}\{\text{acos}[d_{\boldsymbol{p}}(\boldsymbol{p},\boldsymbol{q})] + \text{acos}[d_{\boldsymbol{q}}(\boldsymbol{p},\boldsymbol{q})]\} = 0.120$



(b)

$\boldsymbol{D}'(\boldsymbol{p}) = [0.870, 0.492]$
$\boldsymbol{D}'(\boldsymbol{q}) = [0.473, 0.881]$
$\boldsymbol{L}(\boldsymbol{p},\boldsymbol{q}) = [0.707, -0.707]$
$d_{\boldsymbol{p}}(\boldsymbol{p},\boldsymbol{q}) = 0.267$
$d_{\boldsymbol{q}}(\boldsymbol{p},\boldsymbol{q}) = -0.288$
$f_D(\boldsymbol{p},\boldsymbol{q}) = 0.671$



(c)

$\boldsymbol{D}'(\boldsymbol{p}) = [0.870, 0.492]$
$\boldsymbol{D}'(\boldsymbol{q}) = [-0.881, 0.473]$
$\boldsymbol{L}(\boldsymbol{p},\boldsymbol{q}) = [0.707, -0.707]$
$d_{\boldsymbol{p}}(\boldsymbol{p},\boldsymbol{q}) = 0.267$
$d_{\boldsymbol{q}}(\boldsymbol{p},\boldsymbol{q}) = 0.957$
$f_D(\boldsymbol{p},\boldsymbol{q}) = 0.880$

Figure 4: *Three example computations of f_D: (a) The gradient directions of the two pixels are similar to each other and the link between them, (b) the pixel directions are similar to each other but are near perpendicular to the link between them, and (c) neither the pixel directions nor the link between them are similar.*

of $\boldsymbol{q}$ in $\boldsymbol{p}$'s neighborhood) and point such that the dot product of $\boldsymbol{D}'(\boldsymbol{p})$ and $\boldsymbol{L}(\boldsymbol{p},\boldsymbol{q})$ is positive (i.e., the angle between $\boldsymbol{D}'(\boldsymbol{p})$ and the link $\leq \pi/2$), as noted in (6) above. Figure 4 gives three example computations of f_D. The main purpose of including the neighborhood link direction is to associate a high cost with an edge between two neighboring pixels that have similar gradient directions but are perpendicular, or near perpendicular, to the link between them (Fig. 4(b)). Therefore, the direction feature cost is low when the gradient direction of the two neighboring pixels are similar to each other and the link between them (Fig. 4(a)).

3.1.4 Pixel Value Features (f_P, f_I, f_O)

As mentioned, the pixel value feature costs only have meaning after training. Edge pixel values are simply the scaled source image pixel values directly beneath the portion of the object boundary used for training. Since typical gray-scale image pixel values range from 0 to 255, then the edge pixel value for a pixel $\boldsymbol{p}$ is given by the scaling function

$$f_P(\boldsymbol{p}) = \frac{1}{255} I(\boldsymbol{p}) \tag{7}$$

where $I(\boldsymbol{p})$ is the pixel value of the source image at $\boldsymbol{p}$. The "inside" and "outside" pixel values [12,29] are also taken (and scaled) directly from the source image, but they are sampled at some offset from the defined object boundary. More specifically, the inside pixel value for a given point or pixel $\boldsymbol{p}$ is sampled a distance k from $\boldsymbol{p}$ in the gradient direction and the outside pixel value is

sampled an equal distance in the opposite direction. Thus, the formulation for the inside pixel value, $f_I(\boldsymbol{p})$, and the outside pixel value, $f_O(\boldsymbol{p})$, for a given pixel $\boldsymbol{p}$ is

$$f_I(\boldsymbol{p}) = \frac{1}{255} I(\boldsymbol{p} + k \cdot \boldsymbol{D}(\boldsymbol{p})) \tag{8}$$

and

$$f_O(\boldsymbol{p}) = \frac{1}{255} I(\boldsymbol{p} - k \cdot \boldsymbol{D}(\boldsymbol{p})) \tag{9}$$

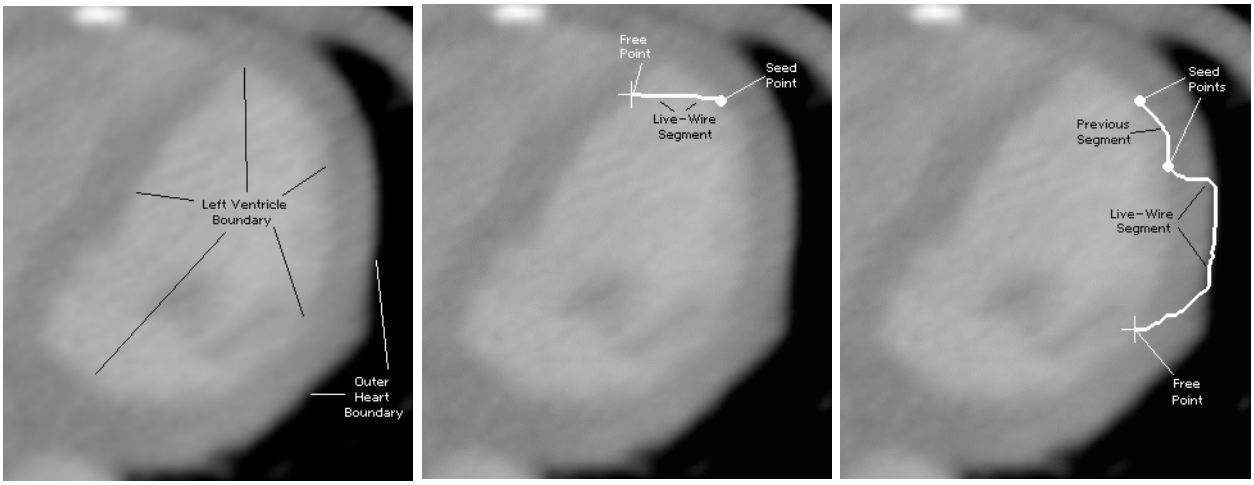
where $\boldsymbol{D}(\boldsymbol{p})$ is the unit vector of the gradient direction as defined in section 3.1.3 and k is either a constant distance value (as determined by the user) or corresponds to a distance 1 pixel larger than half of the optimal kernel width at pixel $\boldsymbol{p}$. Since the resulting sampling position for the inside and outside features will typically not correspond to a pixel's exact center, the value can be taken as the closest pixel (default) or bilinearly interpolated from each of the four surrounding pixels.

3.1.5 Color

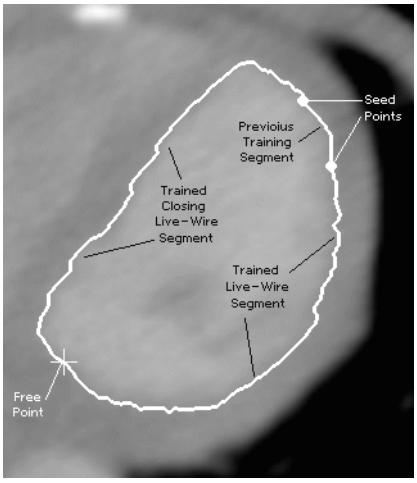
Computing the local cost for color images varies slightly for most of the local cost features. Both the Laplacian zero-crossing and the gradient magnitude are computed by processing each of the three color bands (in RGB color space) independently and combining the results by maximizing over the three respective outputs to produce a single valued local cost image for each feature. Since the Laplacian zero-crossing is a binary feature, a bitwise OR operator achieves the same result as does computing the maximum of the three outputs. The pixel value features, f_P, f_I, and f_O, are currently computed by taking the brightness (in the HSB color space) of the corresponding pixel. The gradient direction computation is unchanged for color images.

3.1.6 On-the-fly Training

Often, an object boundary may not consist of "strong" edge features. For example, Figure 5(a) shows a CT scan of the heart where the boundary of the left ventricle (labelled) has a low gradient magnitude--especially when compared to the much higher gradient magnitude (right of the ventricle) of the heart's nearby outer boundary. Figure 6 compares the histograms of the gradient magnitude values sampled from boundary points on both the left ventricle and outer heart wall and then shows how the static gradient magnitude cost map favors the higher gradient values by mapping them to relatively lower costs. As a result, when trying to track the right boundary of the



(a) (b) (c)



(d)

Figure 5: *(a) CT scan of the heart. (b) Untrained live-wire segment (cuts across corner of left ventricle). (c) Short boundary segment and untrained live-wire segment (snaps to the stronger gradient of the outer heart wall). (d) Live-wire segments (including closing segment) trained on selected boundary segment. Notice that the short training segment is all that is needed to completely define the left ventricle boundary.*

ventricle, the optimal boundary "snaps" to the lower cost outer heart boundary rather than follow the desired higher cost ventricle boundary. Further, since the ventricle boundary's gradient magnitude is relatively low (corresponding to a relatively high static feature cost) then the short, high local cost path that cuts across the upper-left corner of the ventricle produces a cumulative lower cost than the desired longer, slightly lower local cost path around the corner. Both of these problems are resolved when the gradient magnitude feature cost function is determined dynamically from a sample of the desired boundary (static training as applied to boundary finding is introduced in [3]). Figure 6(c) shows a dynamic gradient magnitude cost map created from the histogram of the sampled left ventricle boundary points. Notice how it favors gradient magnitude values similar to those sampled from the left ventricle boundary.

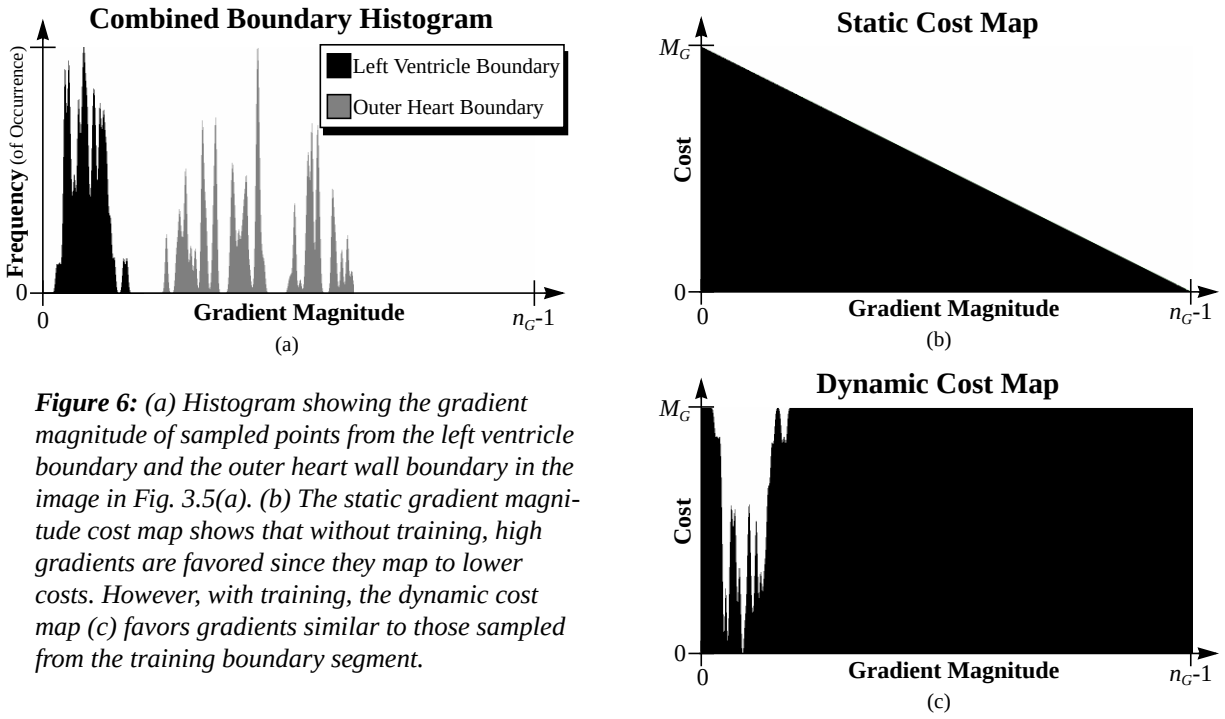


Figure 6: *(a) Histogram showing the gradient magnitude of sampled points from the left ventricle boundary and the outer heart wall boundary in the image in Fig. 3.5(a). (b) The static gradient magnitude cost map shows that without training, high gradients are favored since they map to lower costs. However, with training, the dynamic cost map (c) favors gradients similar to those sampled from the training boundary segment.*

Training allows dynamic adaptation of certain cost feature functions based on a sample boundary segment. Training is performed dynamically as part of the boundary segmentation process. Trained features are updated interactively as an object boundary is being defined. This eliminates a separate training phase and allows the trained feature cost functions to adapt within the object being segmented as well as between objects in the image. Figure 5(d) demonstrates how training was effective in isolating the weaker left ventricle edge to completely define the ventricle's boundary with a single short training segment.

To facilitate sampling of edge characteristics, feature value images are precomputed for all trainable features: the three pixel value features, f_P, f_I, and f_O, and the gradient magnitude feature, f'_G (where $f'_G = G'/\max(G')$ is simply the scaled gradient magnitude). During training, sampled pixel values from these precomputed feature images are used as indices into the corresponding feature histograms. As such, feature value images are computed by simply scaling and rounding f_P, f_I, f_O, and f'_G respectively. Letting I_P, I_I, I_O, and I_G be the feature value images corresponding to the feature cost functions f_P, f_I, f_O, and f'_G, respectively, then

$$
\begin{aligned}
I_P &= \lfloor (n_P - 1) f_P + 0.5 \rfloor \\
I_I &= \lfloor (n_I - 1) f_I + 0.5 \rfloor \\
I_O &= \lfloor (n_O - 1) f_O + 0.5 \rfloor \\
I_G &= \lfloor (n_G - 1) f'_G + 0.5 \rfloor
\end{aligned}
\tag{10}
$$

compute the feature value images where n_P, n_I, n_O = 256 and n_G = 1024 are the respective histogram domains (i.e., number of entries or bins). These feature images are sampled to both create dynamic histograms (which are then scaled, weighted, and inverted to create cost maps) and as indices into the dynamic feature cost maps when computing link costs.

Selection of a “good” boundary segment for training is made interactively using the live-wire tool. To allow training to adapt to gradual (or smooth) changes in edge characteristics, the trained feature cost functions are based only on the most recent or closest portion of the current defined object boundary. A training length or maximum sample size, *t*, specifies how many of the most recent boundary pixels are used to generate the training statistics. A monotonically decreasing weight function, *w,* determines the contribution from each of the closest *t* pixels. The training algorithm samples the precomputed feature value images along the closest *t* pixels of the edge segment and increments the boundary feature histogram element by the corresponding pixel weight to generate a histogram for each feature involved in training.

Since training is based on learned edge characteristics from the most recent portion of an object’s boundary, training is most effective for those objects with edge properties that do not change drastically (or at least change smoothly) as its boundary is traversed. In fact, training can be counter-productive for objects with sudden and/or dramatic changes in edge features. However, the dynamic nature of training allows the user to interactively activate training so that it can be applied to a section of the object boundary and then deactivate it before encountering a sudden transition in edge features.

The training length is typically short (32 to 64 pixels) to allow it to adapt to gradual changes. However, short training segments often result in noisy sampled distributions. Convolving the boundary feature histograms with a 1-D Gaussian helps reduce the effects of noise.

After sampling and smoothing, each feature histogram is then scaled and inverted to create the feature cost map. A maximum local link cost, M, specifies the largest integer cost possible through summation of feature cost components. Each scaled feature's maximum link cost is the product of the feature's weight factor, ω, and the maximum link cost value, M. For example, the maximum gradient magnitude link cost is $M_G = \omega_G \cdot M$. These maximum feature cost values are used as scaling factors when converting the sampled histograms into feature cost maps. Thus, letting h_G represent the sampled and smoothed gradient magnitude histogram, the dynamic gradient magnitude cost map, m_G, is computed by inverting h_G, scaling and rounding as follows:

$$m_G = \left\lfloor \frac{\max(h_G) - h_G}{\max(h_G)} M_G + 0.5 \right\rfloor = \left\lfloor M_G \left(1 - \frac{h_G}{\max(h_G)} \right) + 0.5 \right\rfloor \tag{11}$$

where the division by $\max(h_G)$ scales the histogram between 0 and 1 for further scaling by M_G. The same equation is used for the other dynamic feature cost maps, m_P, m_I, and m_O, with appropriate substitutions of h_P, h_I, and h_O for h_G and M_P, M_I, and M_O for M_G.

Gradient magnitude is the only feature cost that has both static and dynamic (trained) functions. As such, it is often desirable to combine both the static and dynamic functions. One such case arises when t boundary points are not available for sampling. In such a case the sampled distribution is even more noisy and less reliable. To overcome this, a scaling length or minimum sample size, s, determines how many boundary pixels constitute a "reliable" sample. Since a boundary sample containing fewer than s pixels is deemed to contain insufficient data to create a reliable dynamic gradient magnitude cost map, the static gradient cost function is combined with the sampled cost map, m_G. The minimum sample size, s, and the actual number of sampled points, $t_s \le t$ (where t is the training length or maximum sample size specified previously), are used to compute a adjusted static gradient magnitude cost which is then combined with m_G (the gradient magnitude cost map). Thus, the new, combined gradient magnitude cost map, m'_G (combining the static and dynamic gradient magnitude components), is

$$m'_G(x) = \begin{cases} \min\left(m_G(x), \left\lfloor M_G \left[1 - \dfrac{x(s - t_s)}{(n_G - 1)s} \right] + 0.5 \right\rfloor \right); & \text{if } t_s < s \\ m_G(x) \qquad\qquad\qquad\qquad\quad ; & \text{if } t_s \ge s \end{cases} \tag{12}$$

where $x = 0, 1, \ldots, n_G - 1$ is the domain of m'_G, M_G is the gradient magnitude scaling factor, and m_G is the sampled gradient magnitude cost map.

Notice that no restriction was placed on the size of the minimum sample size s in relation to t; thus, if $s > t$ then the inverse linear ramp is always present, though not dominant, in the cost map. Notice further that if $t_s = 0$ (i.e., no training data is available), then m'_G simply produces the ***unadjusted*** static gradient magnitude cost function (an inverse linear ramp). Thus, m'_G computes both the static and dynamic gradient magnitude cost functions.

Finally, given as input a connected sequence of t_S points (i.e., pixel positions), $\boldsymbol{p}_i$ for i=0,1,...,t_S-1 such that $p_i \neq p_{i+1}$ and $\|p_{i+1} - p_i\| \leq \sqrt{2}$, the training algorithm is as follows:

Algorithm 1: *Training on boundary segment.*

```
Input:
   t_S≤t                                  {# of boundary points sampled.}
   p_i for i=0,1,...,t_S-1                {Connected point sequence.}
   σ                                      {Smoothing kernel scale.}

Data Structures:
   w                                      {Training weight vector.}
   h_G,h_P,h_I,h_O                        {Feature histograms.}

Output:
   m'_G,m_P,m_I,m_O                       {Trained feature cost maps.}

Algorithm:
   clear(h_G);                            {Clear all feature histograms.}
   clear(h_P);
   clear(h_I);
   clear(h_O);

   for i=0 to t_S-1 do begin              {Sample feature points.}
      v=I_G(p_i);  h_G(v)=h_G(v)+w(i);
      v=I_P(p_i);  h_P(v)=h_P(v)+w(i);
      v=I_I(p_i);  h_I(v)=h_I(v)+w(i);
      v=I_O(p_i);  h_O(v)=h_O(v)+w(i);
   end

   smooth(h_G,σ);                         {Smooth histograms by σ.}
   smooth(h_P,σ);
   smooth(h_I,σ);
   smooth(h_O,σ);

   m_G=(1-h_G/max(h_G))*M_G               {Scale and invert histograms.}
   m_P=(1-h_P/max(h_P))*M_P
   m_I=(1-h_I/max(h_I))*M_I
   m_O=(1-h_O/max(h_O))*M_O

   if s>t_S then                          {Add in static gradient magnitude map.}
      for v=0 to n_G-1 do
         m'_G(v)=min(m_G(v),floor(M_G*(1-(x*(s-t_S))/((n_G-1)*s))+0.5));
```

3.1.7 Static Neighborhood Link Cost

Since training is not available on the Laplacian zero-crossing and gradient direction features, these costs are precomputed and combined into a static neighborhood cost map, thereby avoiding expensive cost computations within the interactive live-wire environment. These combined costs are computed for every link by summing the scaled, rounded local static cost functions. Given a point, $\boldsymbol{p}$, and any neighboring point, $\boldsymbol{q}$, the static link cost map, l_S, is

$$l_S(\boldsymbol{p}, \boldsymbol{q}) = \lfloor M_Z \cdot f_Z(\boldsymbol{q}) + 0.5 \rfloor + \lfloor M_D \cdot f_D(\boldsymbol{p}, \boldsymbol{q}) + 0.5 \rfloor \tag{13}$$

where M_Z and M_D are the maximum Laplacian zero-crossing and gradient orientation link cost (similar to M_P, M_I, M_O, and M_G defined for Eq. (11)). Since there are 8 neighbors for each pixel, the precomputed static link map, l_S, requires $8N$ cost values for N image pixels.

3.1.8 Final Local Link Cost

Finally, to compensate for differing distances to a pixel's neighbors, gradient magnitude costs are weighted by Euclidean distance. The local gradient magnitude costs to horizontal and vertical neighbors are scaled by $1/\sqrt{2}$ and to diagonal neighbors by 1. Thus, the weighting function w_N for a neighbor $\boldsymbol{q}$ of a pixel $\boldsymbol{p}$ is

$$w_N(\boldsymbol{p}, \boldsymbol{q}) = \begin{cases} 1 ; & \text{if } L_x(\boldsymbol{p}, \boldsymbol{q}) \neq 0 \ \wedge \ L_y(\boldsymbol{p}, \boldsymbol{q}) \neq 0 \\ \dfrac{1}{\sqrt{2}}; & \text{if } L_x(\boldsymbol{p}, \boldsymbol{q}) = 0 \ \vee \ L_y(\boldsymbol{p}, \boldsymbol{q}) = 0 \end{cases} \tag{14}$$

where L_x and L_y are the horizontal and vertical components of the bidirectional link vector $\boldsymbol{L}$ defined in Eq. (6).

As described in Eq. (1), the local cost function, l, is a weighted summation of feature cost functions (f_Z, f_D, f_G, etc.) and ranges from 0 to 1. However, we create an updated local cost function, l', with an integer range between 0 and M - 1 (inclusive) which incorporates training, the precomputed static link map, l_S, and the Euclidean distance weighting function. The resulting, updated local cost function, l', for a neighbor $\boldsymbol{q}$ of a pixel $\boldsymbol{p}$ is

$$l'(\boldsymbol{p}, \boldsymbol{q}) = l_S(\boldsymbol{p}, \boldsymbol{q}) + w_N(\boldsymbol{p}, \boldsymbol{q}) \cdot m'_G(I_G(\boldsymbol{q})) + m_P(I_P(\boldsymbol{q})) + m_I(I_I(\boldsymbol{q})) + m_O(I_O(\boldsymbol{q})) \tag{15}$$

where l_S is the static link cost in Eq. (13), w_N is the neighborhood weighting function in Eq. (14), each m (or m'_G) is the corresponding feature's mapping function generated through training as defined in Eq. (11) (or Eq. (12)), and each I is the precomputed feature value image for the corresponding feature (Eq. (10)).

3.2 Unrestricted Graph Search

Although this work was motivated by [22,23], it does not utilize either the graph formulation or the optimal path computation [28] described therein. Rather, our graph formulation is pixel based rather than crack based and we utilize a more efficient optimal graph search algorithm based on Dijkstra's [11] algorithm. Note that Nilsson's A* algorithm [27], utilized in both [20] and [7], is essentially Dijkstra's algorithm with an additional heuristic which can be used to prune the graph search. This paper extends previous optimal graph search boundary finding methods in 3 ways:

1) It imposes no sampling or searching constraints.
2) The active list is sorted with a specialized O(N) bucket sort (where N is the number of pixels processed in the image).
3) No *a priori* goal nodes/pixels are specified.

First, with the exception of [22,23,28], many of the previous boundary finding techniques that utilize graph searching or dynamic programming impose searching and/or sampling constraints to reduce the problem size and/or enforce specific boundary properties. This paper imposes no such constraints, thereby providing object boundaries with greater degrees of freedom and generality. Second, this paper uses discrete local costs within a range. This permits the use of a specialized bin sort algorithm that inserts points into a sorted list (called the active list) in constant time. Finally, since the live-wire tool determines a goal pixel after the fact, the graph search algorithm must compute the optimal path to all pixels since any one of them may subsequently be chosen--but this is the key to the interactive nature of the live-wire tool.

The graph search algorithm is initialized by placing a start or seed point, ***s***, with a cumulative cost of 0, on an otherwise empty list, L (called the active list). A point, ***p***, is placed on the active list in sorted order based on its total or cumulative cost, $g(\boldsymbol{p})$. All other points in the image are (effectively) initialized with infinite cost[1]. After initialization, the graph search then iteratively generates a minimum cost spanning tree of the image, based on the local cost function, l'. In each iteration, the point or pixel ***p*** with the minimum cumulative cost (i.e., the point at the start of the sorted list) is removed from L and "expanded" by computing the total cost to each of ***p***'s unexpanded neighbors. For each neighbor ***q*** of ***p***, the cumulative cost to ***q*** is the sum of the total cost to ***p*** plus the local link cost from ***p*** to ***q***--that is, $g_{tmp} = g(\boldsymbol{p}) + l'(\boldsymbol{p}, \boldsymbol{q})$. If the newly computed total cost to ***q*** is less than the previous cost (i.e., if $g_{tmp} < g(\boldsymbol{q})$) then $g(\boldsymbol{q})$ is assigned the new, lower cumulative cost and an optimal path pointer is set from ***q*** back to ***p***. After computing the cumulative cost to ***p***'s unexpanded neighbors and setting any necessary optimal path pointers, ***p*** is marked as expanded and the process repeats until all the image pixels have been expanded.

The active list is implemented as an array of sublists where the array size is the range of discrete local costs, M. Each sublist corresponds to points with equal cumulative path cost. As such, the order of points within a sublist is not important and can be arbitrary. Consequently, the sublists are singly linked list implementations of stacks. Let $L(i)\downarrow\boldsymbol{q}$ denote that a point ***q*** with cumulative path cost c is added to the active list in sorted order by pushing ***q*** onto the stack at list array index $i = c \bmod M$. If M is a constant power of 2, the modulo operation can be replaced with a faster bitwise *AND* operation resulting in $i = c$ *AND* $(M - 1)$. Thus, adding a point to the active list requires one bitwise *AND* operation to compute the stack index, the corresponding array indexing operation, and then two pointer assignments to push the point on the stack.

Let $N(\boldsymbol{p})$ be the set of pixels neighboring ***p*** and $e(\boldsymbol{p})$ be a boolean mapping function indicating that a point ***p*** has been expanded. Further, let $ptr(\boldsymbol{q})$ be the optimal path pointer for the point ***q***, then the unrestricted graph search algorithm is as follows:

1. The points are simply marked as not yet having a cumulative cost

Algorithm 2: *Unrestricted graph search.*

```
Input:
   s                  {Start (or seed) point/pixel.}
   l(p,q)             {Local cost function for link between pixels p and q.}

Data Structures:
   L                  {List of active pixels sorted by total cost (initially empty).}
   N(p)               {Neighborhood set of p (contains 8 neighbors of pixel).}
   e(p)               {Boolean function indicating if p has been expanded/processed.}
   g(p)               {cumulative cost function from seed point to p.}

Output:
   ptr                {Pointers from each pixel indicating the minimum cost path.}

Algorithm:
   g(s)=0;  L(0)↓s;   {Initialize active list with zero cost seed point.}
   while L≠∅ do begin {While there are unexpanded points:}
      p←min(L);                              {Remove minimum cost point p from list.}
      e(p)=TRUE;                             {Mark p as expanded (i.e., processed).}

      for each q∈N(p) such that not e(q) do begin
         g_tmp=g(p)+l'(p,q);                 {Compute cumulative cost to neighbor.}
         if q∈L and g_tmp<g(q) then begin    {Remove higher cost neighbor }
            i=g(q) AND (M-1);  q←L(i);       {  from list.}
         end
         if q∉L then begin                   {If neighbor not on list,}
            g(q)=g_tmp;                      {  assign neighbor's cumulative cost,}
            ptr(q)=p;                        {  set (or reset) back pointer,}
            i=g(q) AND (M-1);                {  compute (new) index into list,}
            L(i)↓q;                          {  and place on (or return to) the}
         end                                 {  active list.}
      end
   end
```

This algorithm is implemented twice with different computations for the local link cost $l'(\boldsymbol{p}, \boldsymbol{q})$. The local link cost $l'(\boldsymbol{p}, \boldsymbol{q})$ does not change from the previous definition if training is applied. When training is not active, the local link cost function is

$$l'(\boldsymbol{p}, \boldsymbol{q}) = l_S(\boldsymbol{p}, \boldsymbol{q}) + w_N(\boldsymbol{p}, \boldsymbol{q}) \cdot m'_G(I_G(\boldsymbol{q})) \tag{16}$$

where the gradient magnitude mapping function is simply computing the static inverse linear ramp. Using Eq. (16) when training is off provides better computational efficiency in the interactive live-wire environment.

Removing the next minimum cumulative cost point from the sorted list is denoted by $\boldsymbol{p}\leftarrow min(L)$ and involves searching the array of sublists for the first sublist with at least one point on it. The search begins at the index corresponding to the cumulative cost of the last expanded point and proceeds incrementally, wrapping around to 0 when the end of the array is reached, until it finds a non-empty stack index. Specifically, if c is the cumulative cost of the last expanded point, then removing the next minimum cumulative cost point $\boldsymbol{p}$ from L is given by

```
c=c-1;                  {Decrement c to compensate for increment in loop.}
repeat                  {Search for next lowest cost non-empty stack:}
   c=c+1;               {  Increment c to next highest cumulative cost.}
   i=c AND (M-1);       {  Compute list array index.}
until L(i)≠∅
p↑L(i);                 {Pop next minimum cost point of stack i.}
```

where $\boldsymbol{p}\uparrow L(i)$ denotes popping the point $\boldsymbol{p}$ off of the stack at index i on the list. Obviously, removing the minimum cost point from the sorted list cannot be done in constant time. In the worst case (assuming that L is not empty), the search would require M - 1 iterations to find the next point. However, assuming that a point is added to the list at any index with equal probability, the analogy of a snow plow during a storm can be applied for demonstration. If a plow is clearing a circular path repeatedly during a snow storm, the part of the path with the deepest snow is always just in front of the plow. Likewise, the active points currently on the sorted list should generally be most concentrated at indexes just above the index for the cumulative cost c of the last point expanded.

Notice that since the active list is sorted, when a new, lower cumulative cost is computed for a point already on the list, then that point must be removed from the list and added with the lower cost. $\boldsymbol{q}\leftarrow L(i)$ denotes removing the point $\boldsymbol{q}$ from the stack at index i. Like adding a point to the sorted list, this operation is performed in constant time. Pointers for every pixel keep track of the location of each point on the active list. The stack index for the point is also already known (by keeping the cumulative cost for each pixel). Since the order of points on a sublist is not important, the data for the point being removed is overwritten with the data from the head of the sublist (or top of stack) and the stack is then popped, thereby preventing the need to search for and reassign pointers in the single linked list implementation of the stack.

Figure 7 demonstrates how the graph search algorithm creates a minimum cumulative cost path map (with corresponding optimal path pointers). Figure 7(a) is the initial local cost map with the seed point circled. For simplicity of demonstration the local costs in this example are pixel based rather than link based and can be thought of as representing the gradient magnitude cost feature. Figure 7(b) shows a portion of the cumulative cost and pointer map after expanding the seed point (with a cumulative cost of zero). Notice how the diagonal local costs have been scaled by Euclidean distance (consistent with the gradient magnitude cost feature described previously). Weighting by Euclidean distance demonstrates how the cumulative costs to points currently on the

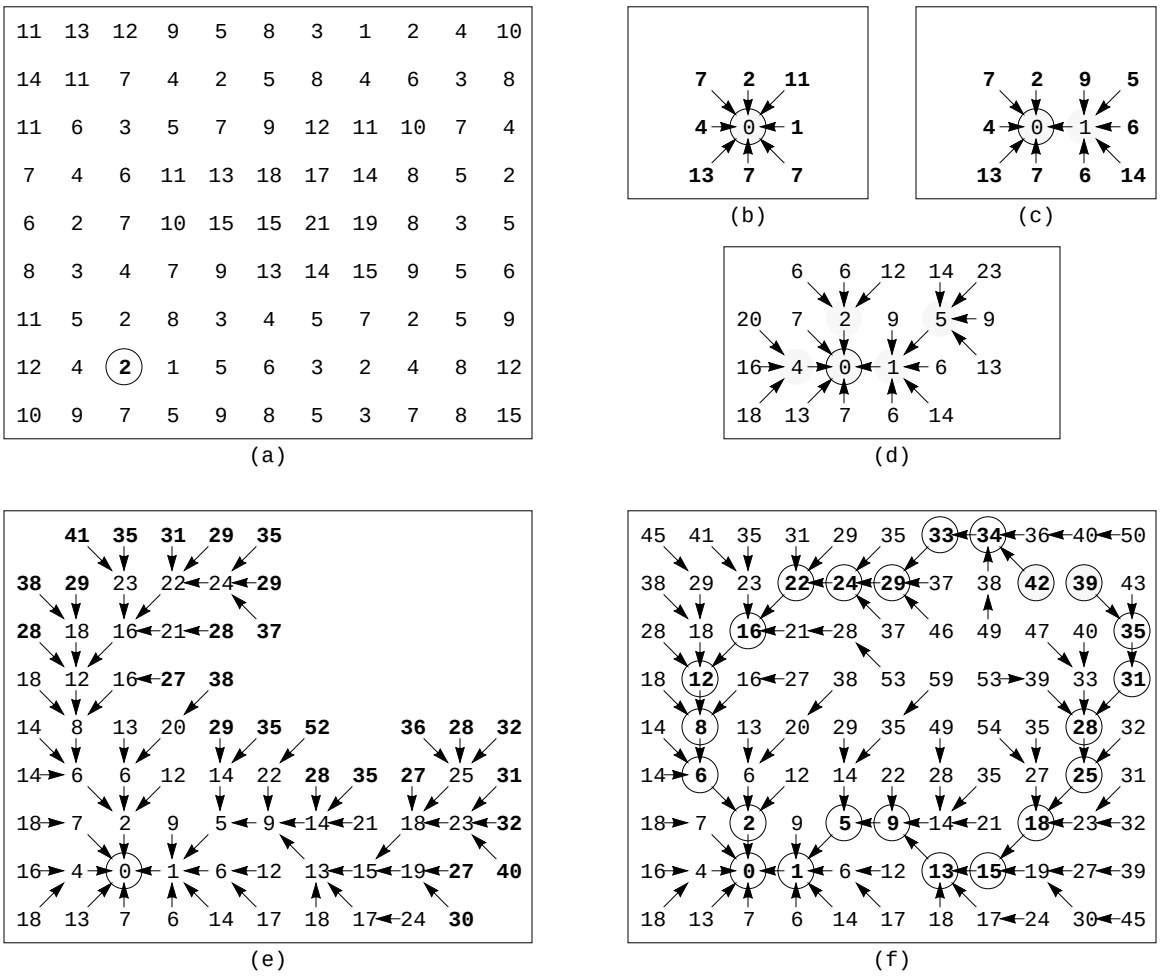


Figure 7: *(a) Initial local cost matrix. (b) Seed point (shaded) expanded. (c) 2 points (shaded) expanded. (d) 5 points (shaded) expanded. (e) 47 points expanded. (f) Finished cumulative cost and path matrix with two of many paths (free points shaded) indicated.*

active list (bold numbers) can change if even lower cumulative costs are computed from as yet unexpanded neighbors. This is demonstrated in Figure 7(c) where two points have now been expanded--the seed point and the next lowest cumulative cost point. Notice how the points diagonal to the seed point have changed cumulative cost and direction pointers. The Euclidean weighting between the seed and diagonal points makes them more expensive than horizontal or vertical paths. Figures 7(d-f) show the cumulative cost/direction pointer map at various stages of completion. Note how the algorithm produces a "wavefront" of active points and that the wavefront grows out faster in areas of lower costs.

3.3 Interactive "Live-Wire"

Once the optimal path pointers are generated, a desired boundary segment can be chosen dynamically via a "free" point. Interactive movement of the free point by the mouse cursor causes the boundary to behave like a live-wire as it adapts to the new minimum cost path by following the optimal path pointers from the free point back to the seed point. Thus, by constraining the seed point and free points to lie near a given edge, the user is able to interactively wrap the live-wire boundary around the object of interest. Figure 8 demonstrates how a live-wire boundary segment adapts to changes in the free point (cursor position) by latching onto more and more of an object boundary. When movement of the free point causes the boundary to digress from the desired object edge, interactive input of a new seed point prior to the point of departure reinitiates the unrestricted graph search expansion. This causes potential paths to be recomputed from the new seed point while effectively "tying off" the boundary computed up to the new seed point.

Since only one optimal path exists from every pixel (or free point) to the seed point, a closed boundary surrounding an object of interest cannot be generated with a single seed point. A minimum of two seed points must be placed to ensure a closed object boundary. The path map from the first seed point of every object is maintained during the course of an object's boundary definition to provide a path from the free point which specifies a closing boundary segment after two or more seed points are specified. The closing boundary segment from the free point to the first seed point eliminates the need for the user to manually close off the boundary.

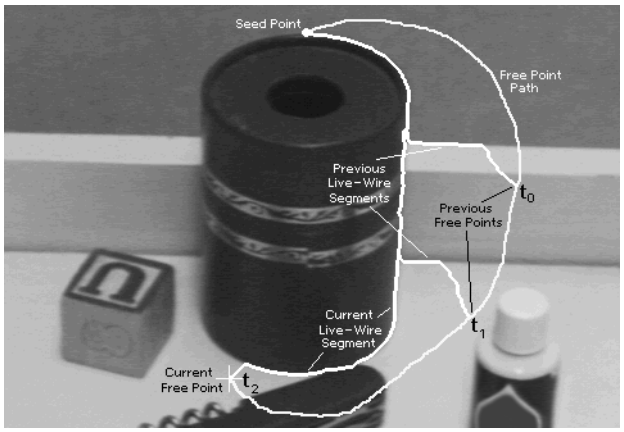
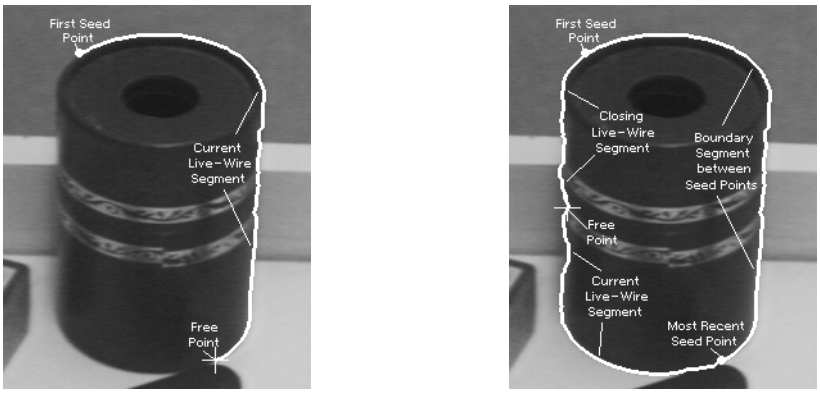


Figure 8: *Example of live-wire snap. As the free point changes via cursor movement, the live-wire segment is updated and displayed from each free point position. The live-wire segments from three different free points and the same seed point are shown (two previous paths from free points at times t_0 and t_1 and the current live-wire path from the free point at time t_2).*



(a) (b)

Figure 9: *(a) With only a single seed point, only a single optimal path is available from a given free point. (b) However, with an additional seed point, both current and closing live-wire segments are specified to create a closed object boundary (in conjunction with the optimal boundary segment between seed points).*

Figure 9 illustrates why a minimum of two seed points are necessary to ensure a closed boundary with the live-wire tool. Fig. 9(a) contains only one seed point and since the free point only specifies a single pixel, only a single optimal path is specified and drawn. However, with a minimum of two seed points, a single free point can specify the optimal path back to the most recent seed point (the current live-wire segment) and the optimal path back to the first seed point placed for that object (the closing segment).

3.3.1 Cursor Snap

Placing seed points directly on an object's edge is often difficult and tedious. If a seed point is not localized to an object edge then spikes results on the segmented boundary at those seed points. To facilitate seed point placement, a cursor snap is available which forces the mouse pointer to the maximum gradient magnitude pixel within a user specified neighborhood [15,25]. The neighborhood can be anywhere from 1×1 (resulting in no cursor snap) to 19×19 (where the cursor can snap as much as 9 pixels in both x and y). Cursor snap is interactively computed for a given neighborhood size by finding, for the current pixel $\boldsymbol{p}$ corresponding to the mouse cursor, the maximum dynamic gradient magnitude pixel, $\boldsymbol{q}$, within $\boldsymbol{p}$'s neighborhood. Thus, as the mouse cursor is moved by the user, the free point immediately snaps or jumps from $\boldsymbol{p}$ to $\boldsymbol{q}$, a neighborhood pixel representing a "good" edge point, thereby facilitating placement of subsequent seed points.

3.3.2 Automatic Seed Point Generation via Path Cooling

While generating closed boundaries around objects of interest can require as few as two seed points, more than two seed points are often required to accurately define an object's boundary. Typically, two to five seed points are required for simple boundary definition but complex objects may require many more. Even with cursor snap, manual placement of seed points can be tedious and often requires a large portion of the overall time required for boundary definition.

Path cooling relieves the user from placing most seed points by automatically selecting a pixel on the current active boundary segment to be a new seed point. Selection is based on "path cooling" which in turn relies on path coalescence. Even though only a single minimum cost path exists from each pixel to a given seed point, many paths "coalesce" and share portions of their optimal path with other paths from other pixels. If any two optimal paths from two distinct pixels share a common point or pixel, then the two paths are identical from that pixel back to the seed point. This is primarily noticeable if the seed point is placed near an object edge and the free point is moved away from the seed point but remains in the vicinity of the object edge. Though a new path is selected and displayed every time the mouse cursor moves, the paths are typically all identical near the seed point and only change local to the free point. As the free point moves farther and farther away from the seed point, the portion of the active "live-wire" boundary segment that does not change becomes longer and longer.

Using boundary cooling, seed points are automatically placed by finding a pixel on the active live-wire segment that has a "stable" history. Each pixel in the image maintains a count in milliseconds of how long it has been included in the active boundary (to estimate time on the active boundary) and also a count of how many times it has been redrawn (to estimate the number of coalesced paths from distinct free points). The time count provides the live-wire segment with a sense of "cooling". The longer a pixel is on a stable section of the live-wire boundary, the more history it accumulates until it eventually "freezes" and automatically produces a new seed point. Figure 10 shows how path cooling facilitates boundary definition by automatically generating seed points for an object boundary.

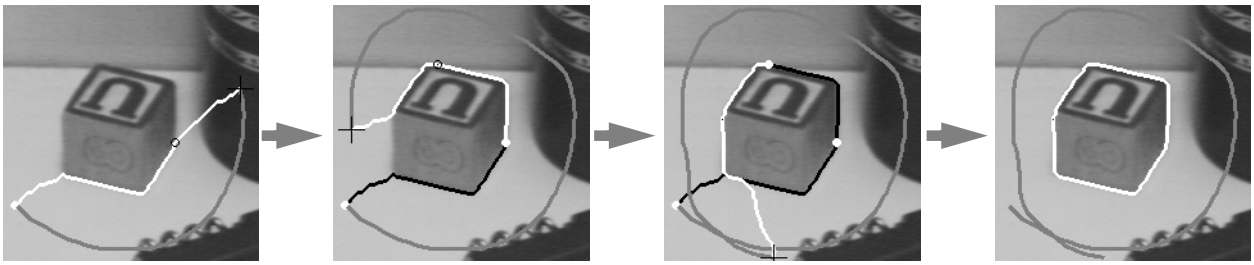


Figure 10: *With path cooling (and overlap detection), the free point or cursor path (shown in gray) is less constrained, allowing the user to simply gesture around the object. As the free point moves, the current live-wire segment (shown in white) cools and freezes, automatically creating a new seed point. The frozen segment turns blue (shown as black) for user feedback.*

The time history is both event and data driven whereas the redraw history is purely event driven. When the free point changes via mouse movement, each pixel's history is updated on the previous live-wire segment by following the pointers from the previous free point back to the seed point and each pixel's history on the segment is updated. Specifically, each pixel's redraw history is incremented and the time history is updated by adding to the time both the number of milliseconds that the segment was displayed and a scaled gradient magnitude value. The gradient magnitude factor is the data driven portion of the time history and causes pixels on strong edge features to cool more quickly than do those that are not on strong edge features.

Both the time and redraw histories have two thresholds: a lower threshold determines if a pixel is a "candidate" for automatic selection and an upper threshold determines if the live-wire segment is "valid". The first pixel with both counts that satisfies the lower thresholds is a ***candidate*** for selection and the *first candidate point* on the live-wire segment containing a pixel that meets both upper thresholds (i.e., a ***valid*** live-wire segment) is chosen as the new seed point.

Ideally, automatic seed points would be placed on the object boundary as far from the last seed point and as close to the current free point as possible. Automatic placement nearer the current free point can be achieved with a single, relatively small threshold for both of the history features, but a small threshold generates seed points close to manually placed points--since a short boundary segment near the manual seed point begins to accumulate a history. Consequently, a single, small threshold will only allow short live-wire segments to be defined before a new seed point is automatically created--thus the motivation for an upper threshold. The lower threshold is rela-

tively small so that *candidate* seed points are close to the current free point and the upper threshold is large so that *valid* live-wire segments are relatively long. Since a relatively large upper threshold produces longer live-wire segments, it effectively "pushes" candidate seed points away from the previous seed point.

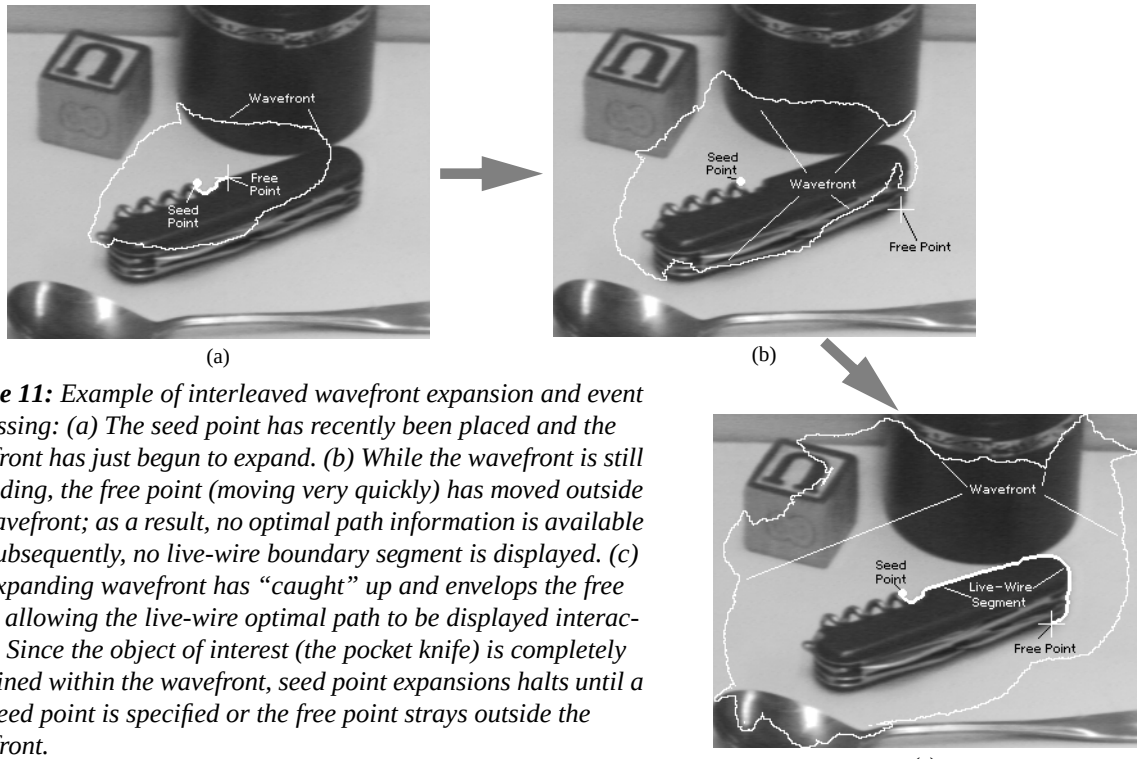
3.3.3 Backup

As with many automatic processes made for very general problem sets, such as arbitrary digital images, seed point generation via path cooling does not always produce seed points on the desired object boundary. Therefore, the cumulative cost and optimal pointer maps (and other corresponding data structures) from old seed points are retained and a backup facility is available if a seed point is placed incorrectly. When backup is specified, the current seed point is removed along with the corresponding cumulative cost map, optimal pointer map, etc., and the previous seed point with its corresponding maps is reinstated as the current active seed point. Backup can be invoked successively to remove several seed points in the reverse order of their creation.

3.3.4 Interleaving Seed Point Expansion with Interactive Live-Wire

Since live-wire segmentation is an interactive tool, delays and lags in processing mouse and other events are undesirable. However, expansion of a seed point to compute optimal paths to every pixel in an image can require several seconds. For a 512x512 image, it requires approximately 2.8 seconds with training (and 1.5 seconds without training) on a 99 MHz HP 735 Unix workstation to compute optimal paths from every pixel in the image to a seed point. Waiting for even one second can become very distracting and counterproductive for such an interactive application, especially with path cooling activated. Interleaving seed point expansion with event processing virtually eliminates lag and results in acceptable interactivity. Since every pixel inside the expanding wavefront already has an optimal path, the goal of interleaving is to keep the wavefront expansion ahead of cursor movement. If the free point moves faster than, and moves outside of, the expanding wavefront then no path is available and therefore no "live-wire" is displayed until the wavefront catches up to the cursor position. For example, Figure 11 shows how the wavefront of pixels on the active list expands out from a selected seed point. In Fig. 11(a), the seed point has

just recently been placed and the wavefront has just begun to expand out into the image. However, even though the wavefront is still expanding, cursor events are processed such that the live-wire boundary is drawn since the cursor position is interior to the wavefront. In Fig. 11(b), the cursor has moved faster than the expanding wavefront, causing the live-wire boundary to temporarily disappear since no optimal path information is available outside the wavefront. In Fig. 11(c), the wavefront has quickly expanded to include the free point; thus, the live-wire optimal path can again be displayed. Typically, the wavefront expansion quickly envelopes an area of interest (i.e., the area of the image where cursor movement is taking place). Consequently, there is rarely any noticeable disappearance of the live-wire segment.



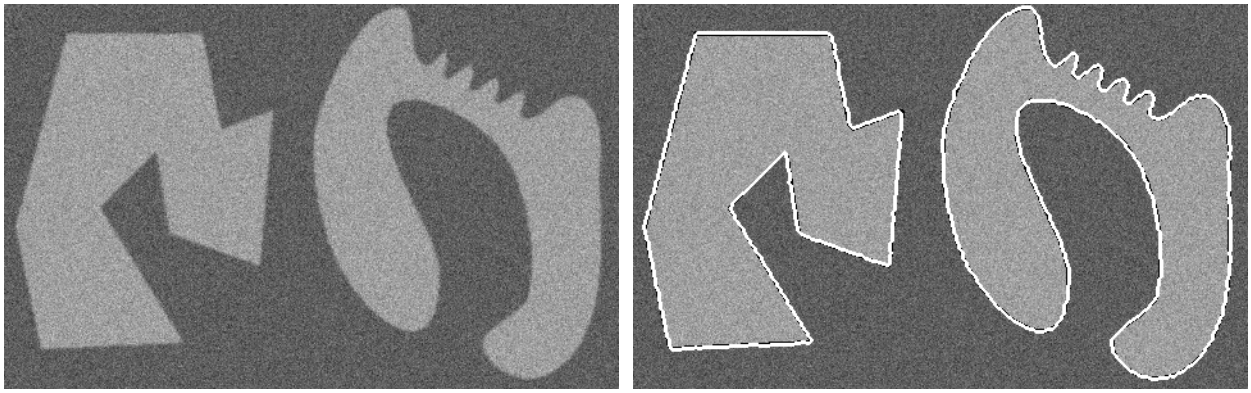
(a) (b) (c)

Figure 11: *Example of interleaved wavefront expansion and event processing: (a) The seed point has recently been placed and the wavefront has just begun to expand. (b) While the wavefront is still expanding, the free point (moving very quickly) has moved outside the wavefront; as a result, no optimal path information is available and subsequently, no live-wire boundary segment is displayed. (c) The expanding wavefront has "caught" up and envelops the free point, allowing the live-wire optimal path to be displayed interactively. Since the object of interest (the pocket knife) is completely contained within the wavefront, seed point expansions halts until a new seed point is specified or the free point strays outside the wavefront.*

4. Results

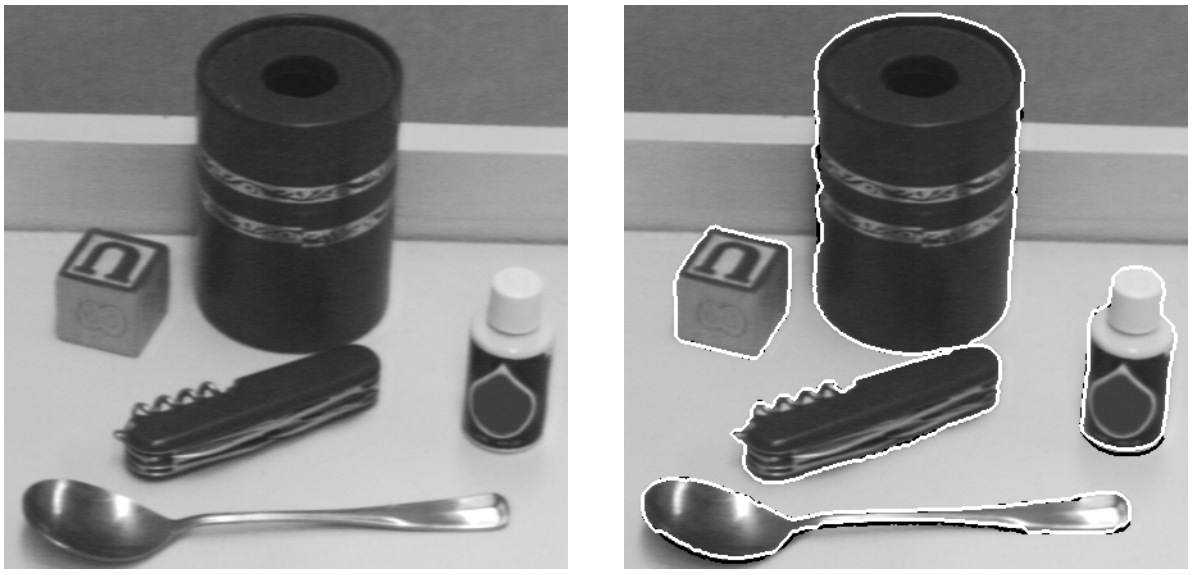
Figures 12 to 18 demonstrate the robustness and generality of Intelligent Scissors on a wide variety of images including real world, color images with complex object boundaries, foregrounds, and backgrounds. In Figures 12 through 15, each live-wire boundary (in white) is overlaid on top of the "ideal" object boundary (in black) for comparison. Thus, black appears only where the live-wire boundary deviates from the ideal. Figure 12 contains a synthetic test image which is used to demonstrate how Intelligent Scissors perform in the presence of edge blurring and white Gaussian noise that may be typical of that produced by a variety of image acquisition hardware. The ideal boundaries in the synthetic image are determined directly from the original binary image. Figures 13 to 15 demonstrate how well live-wire segmentation handles grayscale images acquired with various types of imaging hardware. The ideal boundaries in these images are meticulously defined and, since there is no direct binary standard for comparison, are necessarily subjective. Finally, Figures 16 through 18 demonstrate Intelligent Scissors utility in defining complex real world object boundaries in nontrivial scenes.

Figure 12(a) is a synthetic binary image where the different shapes are created to test the live-wire's ability to track both curved shapes with varying degrees of curvature (note the comb pattern in the right object) and polygonal shapes with sharp corners. Gaussian noise and blur are added to simulate real world images. The boundary definition times for the polygon and curve in Fig. 12 are 4.3 seconds and 8.3 seconds, respectively.



(a) (b)

Figure 12: *(a) Synthetic test image created from a two-color (binary) image by applying Gaussian blur (σ=1.33 pixels) and white, Gaussian noise (σ=16 gray levels). (b) Overlaid "ideal" (black) and live-wire (white) boundary*



(a) (b)

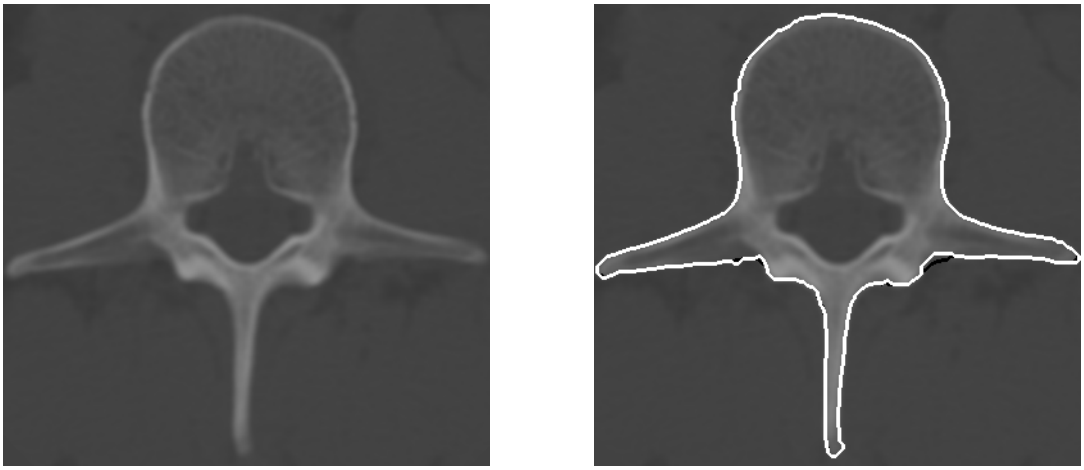
Figure 13: *(a) Desktop image with various objects. (b) Overlaid "ideal" (black) and live-wire (white) boundaries.*

Fig. 13 is an arranged "desktop" scene that, by design, contains difficulties for typical local edge following algorithms, such as where the pocket knife and the paper-clip holder touch. Figure 13(b) overlays the manually defined "ideal" boundaries with the live-wire boundaries to demonstrate how closely the live-wire boundaries match the ideal. The actual times[2] required and number of seed points needed to define each object boundary are as follows:

Desktop Object	**Time** (in seconds)	**# of Seed Points**
Paper Clip Holder	3.6	2
Block	2.4	2
Pocket Knife	4.6	4
Correction Fluid	5.1	4
Spoon	9.8	8

Figures 14 and 15 demonstrate the live-wire's functionality on medical images. Fig. 14 is a CT scan of a spinal vertebrae. The outer boundary of the vertebrae required 5.9 seconds and 5 seed points for live-wire boundary definition. Notice in Fig 14(b) that the live-wire boundary varies noticeably from the "ideal" boundary just right and slightly down of center although the white, live-wire boundary appears more correct in that area of the vertebrae.

2. Times given are for the actual boundaries presented and represent the best time (from several boundary definition attempts) for an acceptable boundary.



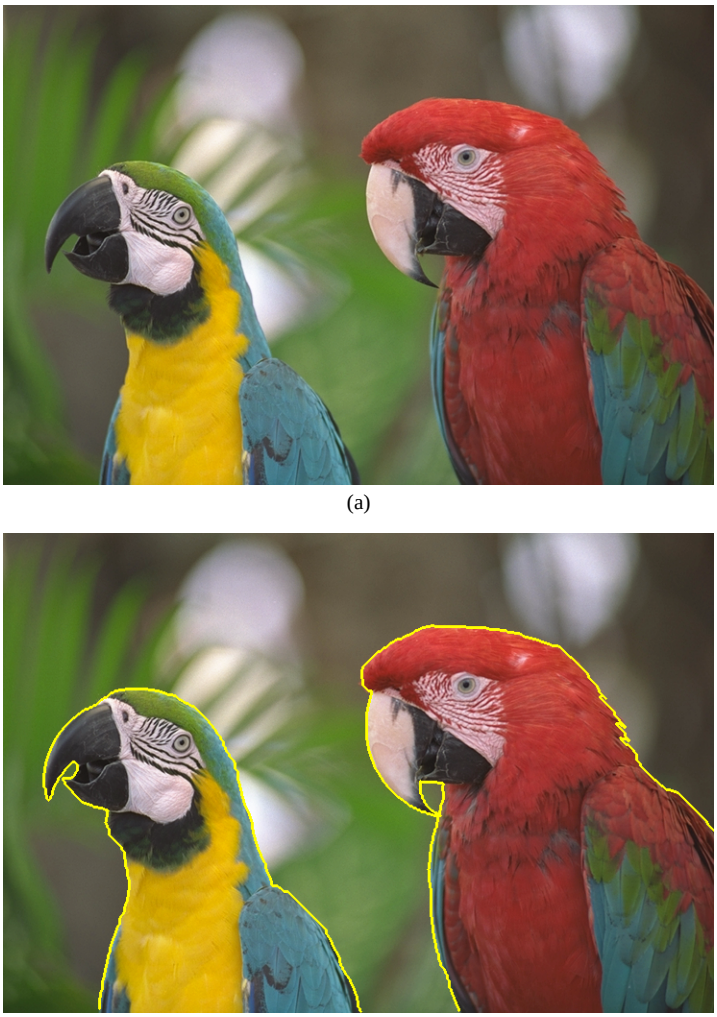
(a) (b)

Figure 14: *(a) CT scan of spinal vertebrae. (b) Overlaid "ideal" (black) and live-wire (white) boundary.*

Figure 15(a) shows an angiogram of a coronary vessel. The left boundary is defined in 2.6 seconds with 3 seed points whereas the right side is defined with only a single seed point/free point pair in 1.9 seconds. As can be seen, the live-wire boundaries agree well with the "ideal" boundaries.

(a) (b)

Figure 15: *(a) Coronary angiogram of coronary artery. (b) Overlaid "ideal" (black) and live-wire (white) boundary.*

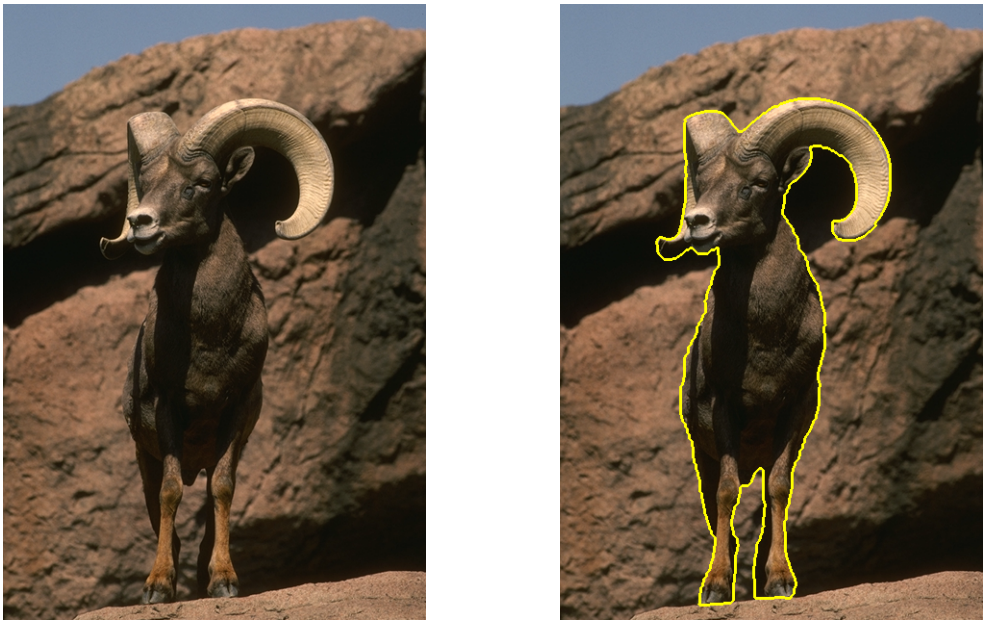


(a)

(b)

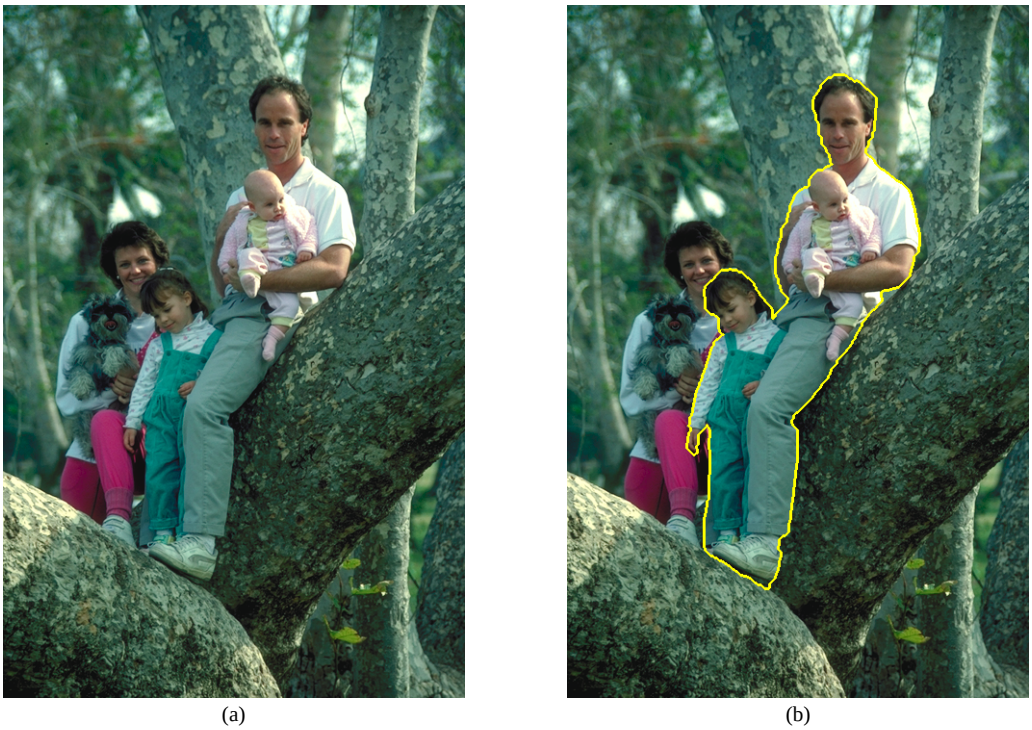
Figure 16: *(a) Image of parrots with nonhomogeneous regions for both the foreground and background. (b) Resulting live-wire boundary (in yellow) using Intelligent Scissors required 16.1 seconds (866 boundary pixels) for the left bird and 17.9 seconds (902 boundary pixels) for the right bird (Image size: 740×500).*

Figures 16 through 18 are full color images which demonstrate Intelligent Scissors' generality and application to complex, real world scenes and object boundaries. The object boundaries in these images are not trivial and demonstrate the power and diversity of live-wire segmentation. As with the horse in Figure 1, the object boundaries in Figures 16 thought 18 contain areas of strong, well isolated edge features which can be defined with long live-wire segments while other areas of the same objects require more human guidance (and thereby shorter live-wire segments) to specify the desired object boundary and isolate it from nearby edge information in either the background or the foreground. Due to the interactive optimal path selection inherent in Intelligent Scissors, the user is able to provide only as much guidance as is necessary to define an object boundary.



(a) (b)

Figure 17: *(a) Image of bighorn sheep that contains regions with similar color as that of the background. (b) Resulting live-wire boundary (in yellow) using Intelligent Scissors required 46.3seconds (1883 boundary pixels). (Image size: 440×640)*



(a) (b)

Figure 18: *(a) Image of a family sitting in a tree. (b) Resulting live-wire boundary (in yellow) using Intelligent Scissors required 23.2 seconds (1349 boundary pixels). (Image size: 480×720)*

4.1 Timing, Accuracy, and Reproducibility

Tables 1 through 4 presents the timing, accuracy and reproducibility (both intra- and inter-user) for the live-wire segmentation tool and compare it with manual tracing of the same objects. These results measure the average accuracy and intra-user variability for 8 different users and tabulate the inter-user variability between them as well. Each user spent some time becoming familiar with the live-wire tool and its interface as well as the manual tracing tool. After they felt comfortable using the two tools, they were asked to manually trace 5 objects 3 times and "live-wire" the same 5 objects 5 times. The five objects are the polygon and the curved shape in the synthetic test image in Fig. 12(a), the paper clip holder and the pocket knife in the desktop image of Fig. 13(a), and finally the outer boundary of the spinal vertebrae in Fig. 14(a).

Table 1 presents the average boundary definition time for manually traced and live-wire boundaries across all users for each object and Table 2 gives the average boundary accuracy for manual and live-wire boundaries across all users for each object. For objects from the synthetic image (where the object boundaries are objectively known), the hand-traced and live-wired boundaries are compared against a Euclidean distance map created directly from the synthetic image's original binary image. Boundaries of real world objects are compared against the distance map created from the "ideal" boundary for each corresponding object. Comparison with the distance map is performed on a pixel by pixel basis and the statistics are accumulated over all boundaries for a given object (and segmentation technique) and over all users.

As can be seen from Tables 1 and 2, the live-wire segmentation performed better in all cases in terms of time required to define the boundary, mean distance and standard deviation from the boundary, and the percent of boundary pixels that are less than 1 pixel and 4 pixels away from the ideal boundary. Since the manual tracing tool can easily define straight line segments, then

Object	**Live-Wire** (in seconds)	**Manual Trace** (in seconds)
Polygon	11.7	20.9
Curve	25.2	66.3
Paper clip holder	10.0	23.5
Pocket Knife	13.7	34.8
Spinal Vertebrae	17.0	49.1

Table 1: *Average timing comparison between live-wire and manually traced boundaries (all users).*

Object	Measure	Mean Distance (pixels)	Standard Deviation (pixels)	Percent Exact	Percent ≤ 1 Pixel	Percent ≤ 4 Pixels
Polygon	Live-Wire	0.018	0.161	97.55	99.78	99.98
	Manual Trace	0.333	0.649	69.14	92.33	99.73
Curve	Live-Wire	0.026	0.311	97.12	99.68	99.88
	Manual Trace	0.862	1.235	44.62	73.53	97.68
Paper clip holder	Live-Wire	0.504	1.438	82.90	88.99	95.40
	Manual Trace	1.505	1.257	19.79	55.30	96.11
Pocket knife	Live-Wire	0.244	0.716	83.72	93.30	99.58
	Manual Trace	1.661	1.314	17.17	48.05	97.38
Spinal vertebrae	Live-Wire	0.203	0.861	91.63	95.33	98.15
	Manual Trace	1.608	1.368	19.77	52.70	94.65
All objects	Live-Wire	0.172	0.806	92.09	96.15	98.67
	Manual Trace	1.157	1.304	35.42	65.66	96.71

Table 2: *Accuracy comparison between live-wire boundaries and hand-traced boundaries.*

objects with straight segments (such as the polygon) are conceptually easier to define than objects with curved edges. But even in the case of the polygon, the hand-traced boundaries required, on the average, almost twice as long to define, and with lower accuracy. Figure 19 is a graphical comparison of the average boundary definition times for each object using live-wire and manual tracing while Figure 20 graphs the difference in accuracy between live-wire and hand-traced boundaries.

Object	Measure	Mean Distance (pixels)	Standard Deviation (pixels)	Percent Exact	Percent ≤ 1 Pixel	Percent ≤ 4 Pixels
Polygon	Live-Wire	0.040	0.280	97.05	99.35	99.91
	Manual Trace	0.916	1.006	35.88	78.47	98.63
Curve	Live-Wire	0.058	0.451	96.34	99.28	99.75
	Manual Trace	1.671	1.664	20.28	53.26	92.85
Paper clip holder	Live-Wire	0.194	0.945	92.63	96.54	98.45
	Manual Trace	1.311	1.328	24.80	66.20	96.73
Pocket knife	Live-Wire	0.118	0.632	93.30	97.83	99.49
	Manual Trace	1.359	1.288	25.83	60.72	95.94
Spinal vertebrae	Live-Wire	0.079	0.588	95.89	98.81	99.61
	Manual Trace	1.700	1.512	19.32	51.67	92.84
All objects	Live-Wire	0.114	0.594	95.46	98.58	99.50
	Manual Trace	1.455	1.461	24.10	60.13	94.82

Table 3: *Intra-user reproducibility comparison between live-wire boundaries and manually traced boundaries.*

Object	Measure	Mean Distance (pixels)	Standard Deviation (pixels)	Percent Exact	Percent ≤ 1 Pixel	Percent ≤ 4 Pixels
Polygon	Live-Wire	0.046	0.295	96.61	99.25	99.91
	Manual Trace	1.060	0.977	28.70	72.15	98.76
Curve	Live-Wire	0.059	0.467	96.37	99.26	99.73
	Manual Trace	1.862	1.683	17.99	47.32	90.44
Paper clip holder	Live-Wire	0.307	1.204	90.19	94.24	96.88
	Manual Trace	1.416	1.368	24.84	62.00	94.79
Pocket knife	Live-Wire	0.144	0.6983	92.22	97.17	99.37
	Manual Trace	1.631	1.436	19.60	51.44	93.66
Spinal vertebrae	Live-Wire	0.087	0.634	95.63	98.68	99.56
	Manual Trace	1.990	1.634	16.30	43.43	89.75
All objects	Live-Wire	0.114	0.700	94.79	8.06	99.20
	Manual Trace	1.676	1.539	20.50	52.98	92.59

Table 4: *Inter-user reproducibility comparison between live-wire boundaries and manually traced boundaries.*

Tables 3 and 4 summarize the overall intra- and inter-user reproducibility results respectively. To compute the intra-user reproducibility for a given user and boundary definition technique (either live-wire or hand-traced), each boundary defined by the user is compared to every other boundary defined for the same object by the same user. These statistics are gathered for each object over all users to produce the overall, average intra-user reproducibility statistics. For the inter-user statistics, each boundary defined with a particular tool (i.e., live-wire or manually traced) for every user is compared against every boundary for the same object defined by every other user using the same tool. Figure 21 is a graph comparing the intra- and inter-user reproducibility between live-wire and hand-traced boundaries. Notice that even the live-wire ***inter***-user reproducibility is considerably better than the ***intra***-user reproducibility for manually traced boundaries.

4.2 Computational Complexity

Previous graph searching/dynamic programming approaches to boundary detection were typically computationally expensive. However, by restricting the local costs to integer values within a range, the graph searching algorithm can take advantage of an O(N) bucket sort where N is the number of image pixels for which optimal paths have been computed from the pixel to a

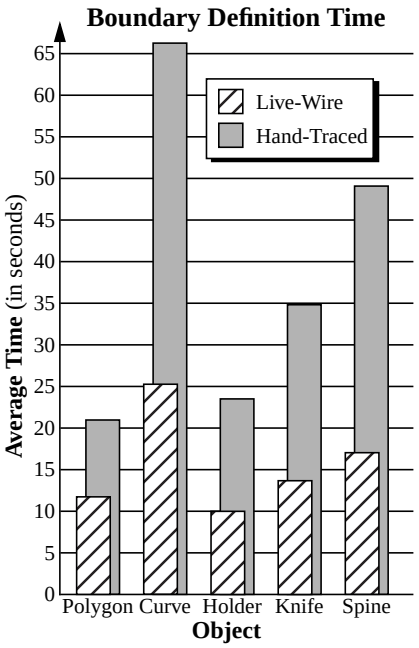


Figure 19: *Graphical comparison between live-wire and hand-traced boundary definition times.*

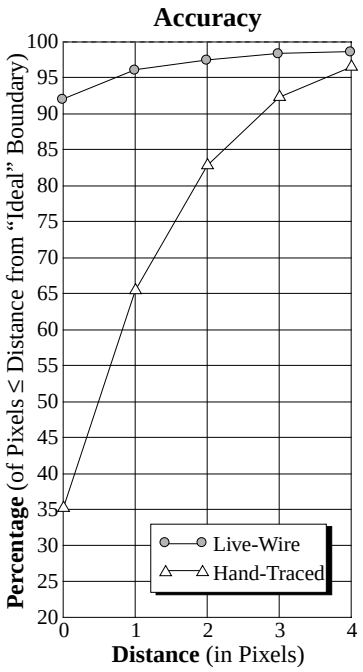


Figure 20: *Graphical comparison between live-wire and hand-traced boundary accuracy.*

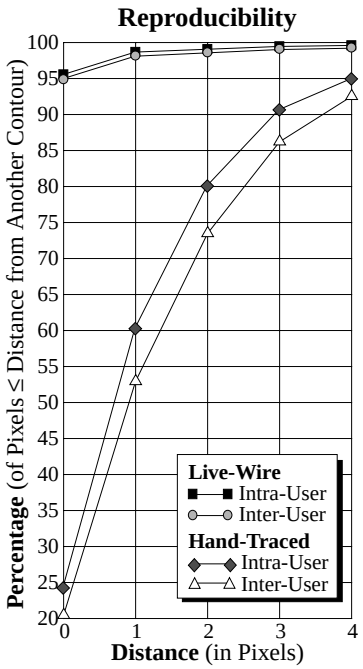


Figure 21: *Graphical comparison between live-wire and hand-traced boundary reproducibility (both intra- and inter-user).*

seed point. As mentioned in Section 3.2, adding points to the sorted active list requires constant time and removing points requires near constant time. As a result, the algorithm's computational complexity[3] for N image pixels is $O(N)$. This can be seen by examining the algorithm in a worst case situation. As a pixel is removed from the active list, it is expanded by computing the cumulative cost to all of its neighbors that have not already been expanded. In the worst case, a pixel computes a cumulative cost to all of its 8 neighbors, resulting in $8N$ cumulative cost computations for N pixels. However, not every point can be expanded after all of its neighbors have. Except for the seed point, every point that has a cumulative cost must have at least one neighbor that has already been expanded. In fact, once an edge between two pixels has been used to compute the cumulative cost as a result of expanding one of the two pixels, that edge will not be considered again for a cumulative cost computation. Thus, each edge between pixels is considered only once. Since each pixel has an edge to each of its 8 neighbors and since each edge is shared by 2 pixels

3. If the wavefront expands to fill an entire $n \times m$ image, then $N = nm$; otherwise, $N < nm$.

then the total number of cumulative cost computations for N pixels is $\frac{8}{2}N = 4N$, which is still O(N). Consequently, the graph expansion, when interleaved with interactive event processing, can compute optimal paths at interactive speeds. As mentioned previously, an HP 735/99 workstation requires approximately 1.5 seconds to compute untrained optimal paths (using Eq. (16)) from a seed point to every other pixel in a 512x512 image whereas with training (using Eq. (15)) it requires approximately 2.8 seconds.

5. Conclusion and Future Work

This paper has presented an interactive image segmentation tool based on an unrestricted graph search. The major contributions of this work are:

1) The addition of the Laplacian zero-crossing binary feature cost that improves edge localization for optimal boundary segments.
2) Due to the ability to add and remove nodes to and from the active list in constant time, this algorithm is less computationally expensive than traditional graph searching/ dynamic programming based boundary finding approaches.
3) The improved computational speed makes interaction possible during optimal path generation, allowing for interactive selection of the desired optimal boundary segment via the live-wire segmentation tool.
4) Cooling helps reduce the need for user input and thereby facilitates and improves the live-wire's interactivity.
5) On-the-fly training is unique from traditional training algorithms that have been applied to boundary definition and allows for training information to be updated and used dynamically as part of the normal boundary definition process.

It is important to note that the last three contributions are realized only through the second contribution: the ability to generate all the optimal paths at interactive speeds.

In conclusion, when compared to tedious manual tracing, the Intelligent Scissors segmentation tool provides a quicker, more accurate, and more reproducible general purpose tool for defining object boundaries within images. As such, Intelligent Scissors can, and has been, applied to medical image volume segmentation, digital image composition, general color and grayscale image segmentation, and line extraction from scanned documents.

5.1 Future Work

Although Intelligent Scissors have dramatically decreased the time and increased the accuracy and reproducibility with which boundaries can be extracted, there are opportunities for extension of this work. Possible additions to the current Intelligent Scissors tool include:

1) Improved training by automatic weight adjustment and feature selection.
2) Subpixel estimation of boundaries for antialiasing.
3) Extending the domain to temporal or spatial sequences of images.

First, training currently relies on feature cost weights that are set (via command line arguments or default values) when the program is initiated. However, it is likely that the reliability or importance of the different feature costs change from one object to another. For that matter, each feature's strength may change within an object's boundary. Since training gathers statistics on various image features, it is possible to adjust feature cost weights based on the variance, standard deviation, or some other similar measure of the feature distribution. A strong feature would tend to exhibit a tight clustering (resulting in a low variance) whereas a weak feature would likely produce a spread or multi-modal distribution. By adjusting the feature weights based on the feature distributions, the live-wire may be better able to adapt to the current object's edge features.

Second, the live-wire tool currently creates a single pixel wide object boundary and, for closed boundaries, assumes that the pixels within the boundary belong to some object of interest. However, it does not specify if the boundary pixels are themselves part of the segmented object or not. In fact, it will often be the case that a boundary pixel cannot be classified as simply belonging to the object or not, rather, a boundary pixel will partially belong to the segmented object and partially belong to the background or some other object. Thus, future research may explore sub-pixel representations for live-wire boundaries.

Finally, applications such as medical volume imaging and image composition for special effects in movies need to segment an object (or group of objects) from each 2-D image plane or frame of a spatial or temporal image sequence. Further work in live-wire segmentation could include extending the tool to segment objects in 3-space where the third dimension is either spatial or temporal. As such, the local costs may be computed using 3-D convolution kernels and the current live-wire tool may be extended to a live-wire in 3-space (as opposed to the current restriction in a 2-D plane) or possibly even to a live-surface tool where an optimal surface segment is selected from a large set of optimal surfaces.

As can be seen, though Intelligent Scissors serve as a useful a general purpose tool for segmenting 2-D object boundaries from images of arbitrary content and complexity, there are still several research areas that could extend and enhance the possibilities of this tool. As such, Intelligent Scissors promise to remain on the *cutting edge* of interactive image segmentation techniques.

6. References

[1] A. A. Amini, T. E. Weymouth, and R. C. Jain, "Using Dynamic Programming for Solving Variational Problems in Vision," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 12, No. 9, pp. 855-866, Sept. 1990.

[2] D. H. Ballard and J. Sklansky, "Tumor Detection in Radiographs", *Computers and Biomedical Research*, Vol. 6, No. 4, pp. 299-321, Aug. 1973

[3] W. A. Barrett, P. D. Clayton, and H. R. Warner, "Determination of Left Vetricular Contours: A Probabilistic Algorithm Derived from Angiographic Images," *Computers and Biomedical Researc*h, Vol. 13, No. 6, pp. 522-548, Dec. 1980.

[4] W. A. Barrett, Personal communication to J. K. Udupa regarding interactive live-wire optimal path selection, Feb. 1992.

[5] W. A. Barrett and E. N. Mortensen, "Fast, Accurate, and Reproducible Live-Wire Boundary Extraction," in *Proceedings of Visualization in Biomedical Computing 96*, pp. 183-192, Hamburg, Germany, Sept. 1996.

[6] W. A. Barrett and E. N. Mortensen, "Interactive Live-Wire Boundary Extraction," *Medical Image Analysis*, Vol. 1, No. 4, pp. 331-341, 1997.

[7] J. D. Cappelletti and A. Rosenfeld, "Three-Dimensional Boundary Following," *Computer Vision, Graphics, and Image Processing*, Vol. 48, No. 1, pp. 80-92, Oct. 1989.

[8] Y. P. Chien and K. S. Fu, "A Decision Function Method for Boundary Detection" *Computer Graphics and Image Processing*, Vol. 3, No. 2, pp. 125-140, June 1974.

[9] L. D. Cohen and R. Kimmel, "Global Minimum for Active Contour Models: A Minimum Path Approach," in *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR '96)*, San Francisco, CA, June 1996.

[10] D. Daneels, et al., "Interactive Outlining: An Improved Approach Using Active Contours," in *SPIE Proceedings of Storage and Retrieval for Image and Video Databases*, Vol. 1908, pp. 226-233, San Jose, CA, Feb. 1993.

[11] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, Vol. 1, pp. 269-270, 1959.

[12] A. X. Falcão, J. K. Udupa, S. Samarasekera, and B. E. Hirsch, "User-Steered Image Boundary Segmentation," in *Proceedings of the SPIE--Medical Imaging 1996: Image Processing*, Vol. 2710, pp. 278-288, Newport Beach, CA, Feb. 1996.

[13] M. M. Fleck, "Multiple Widths Yield Reliable Finite Differences," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 14, No. 4, pp. 412-429, April 1992.

[14] D. Geiger, A. Gupta, L. A. Costa, and J. Vlontzos, "Dynamic Programming for Detecting, Tracking, and Matching Deformable Contours," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 17, No. 3, pp. 294-302, Mar. 1995 (Correction in *PAMI*, Vol. 18, No. 5, pg. 575, May 1996)

[15] M. Gleicher, "Image Snapping," in *Proceedings of the ACM SIGGRAPH 95: 22nd International Conference on Computer Graphics and Interactive Techniques*, pp. 183-190, Los Angeles, CA, Aug. 1995.

[16] H. Jeong and C. I. Kim, "Adaptive Determination of Filter Scales for Edge Detection." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 14, No. 5, pp. 579- 585, May 1992.

[17] M. Kass, A. Witkin, and D. Terzopoulos, “Snakes: Active Contour Models,” in *Proceedings of the First International Conference on Computer Vision*, pp. 259-268, London, England, June 1987.

[18] M. Kass, A. Witkin, and D. Terzopoulos, “Snakes: Active Contour Models,” *International Journal of Computer Vision*, Vol. 1, No. 4, pp. 321-331, Jan. 1988.

[19] D. Marr and E. Hildreth, “Theory of Edge Detection,” *Proceedings of the Royal Society of London--Series B: Biological Sciences*, Vol. 207, No. 1167, pp. 187-217, Feb. 29, 1980.

[20] A. Martelli, “An Application of Heuristic Search Methods to Edge and Contour Detection,” *Communications of the ACM*, Vol. 19, No. 2, pp. 73-83, Feb. 1976.

[21] U. Montanari, “On the Optimal Detection of Curves in Noisy Pictures,” *Communication of the ACM*, Vol. 14, No. 5, pp. 335-345, May 1971.

[22] B. S. Morse, *Trainable Automated Boundary Tracking Using Two-Dimensional Graph Searching with Dynamic Programming*. Masters Thesis, Department of Computer Science, Brigham Young University, Provo, UT, Aug. 1990.

[23] B. S. Morse, W. A. Barrett, J. K. Udupa, and R. P. Burton, *Trainable Optimal Boundary Finding Using Two-Dimensional Dynamic Programming*. Technical Report No. MIPG180, Department of Radiology, University of Pennsylvania, Philadelphia, PA, March 1991.

[24] E. N. Mortensen, B. S. Morse, W. A. Barrett, and J. K. Udupa, “Adaptive Boundary Detection Using 'Live-Wire' Two-Dimensional Dynamic Programming,” in *IEEE Proceedings of Computers in Cardiology*, pp. 635-638, Durham, NC, Oct. 1992.

[25] E. N. Mortensen and W. A. Barrett, “Intelligent Scissors for Image Composition,” in *Proceedings of the ACM SIGGRAPH 95: 22nd International Conference on Computer Graphics and Interactive Techniques*, pp. 191-198, Los Angeles, CA, Aug. 1995.

[26] E. N. Mortensen, *Adaptive Boundary Detection Using 'Live-Wire' Two-Dimensional Dynamic Programmin*g. Masters Thesis, Department of Computer Science, Brigham Young University, Provo, UT, Aug. 1995.

[27] N. J. Nilsson, *Principles of Artificial Intelligence*. Palo Alto, CA: Tioga, 1980.

[28] J. K. Udupa, Personal communication to W. A. Barrett regarding two-dimensional boundary detection using dynamic programming with graph searching. 1989.

[29] J. K. Udupa, S. Samarasekera, and W. A. Barrett, “Boundary Detection via Dynamic Programming,” in *Proceedings of the SPIE: Visualization in Biomedical Computing 92*, Vol. 1808, pp. 33-39, Chapel Hill, NC, Oct. 1992.

[30] D. J. Williams and M. Shah, “A Fast Algorithm for Active Contours and Curvature Estimation,” *CVGIP: Image Understanding*, Vol. 55, No. 1, pp. 14-26, Jan. 1992.