Segmentation of Abnormal Region from Endoscopic Images using Intelligent Scissors

Ravindra S. Hegadi
Department of Computer Science
Karnatak University
Dharwad, India

Shailaja S. Halli
Department of Computer Science
Karnatak University
Dharwad, India

Arpana Kop
Department of Computer Science
Karnatak University
Dharwad, India

ABSTRACT
The commonly found abnormalities in endoscopic images are cancer tumors, ulcers, bleeding due to internal injuries, etc. Several methods of segmentation are employed in recent past for proper segmentation of such images. Intelligent scissors is one of the tools for segmentation. Here the segmentation of endoscopic images is presented using the intelligent scissors. This method is used to segment the tumor, abnormal regions and cancerous growth in the human esophagus. Several methods implemented in the recent past have yielded good results. But this method is simpler. Here a seed point is selected then the cost matrix is constructed which gives the costs of the neighbouring points. Using Dijkstra's algorithm, the nearest point falling in the same region is selected. The proposed method has shown encouraging results in segmenting the abnormal parts from esophegal endoscopic images.

General Terms
Image processing, medical image analysis.

Keywords
Intelligent Scissors, Endoscopy, Segmentation, Optimal Path.

1. INTRODUCTION
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. , 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 [1] in which the boundary is represented as an optimal path within the graph.

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 neighbourhood. 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.

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, 3, 4, 5 and 6]. They formulate the boundary finding problem as a directed graph or cost functional to which an optimal solution is sought. Montanari [7] 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 [8] 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 tumours in chest radiographs. Chien and Fu [9] 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 [10] shows that any optimization problem using dynamic programming can be formulated as a shortest or minimum cost path graph search. He applies Nilsson's [11] 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.

Udupa [12] formulates optimal two-dimensional boundary finding as a graph search using dynamic programming. Like Martelli [10], he formulates the image grid as a directed graph where pixel corners are nodes and the cracks between pixels are directed arcs. Based on Udupa's formulation, and in collaboration with him, Morse et al. [13, 14] 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.

Snakes, active contours, and thin plate models are another global boundary based segmentation techniques that have received a great deal of attention [15, 16, 17, 18, 19, 20, and 21]. 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 energy functional. Kass, Witkin, and Terzopoulos [19, 20] 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 Geiger et al. [18] apply dynamic programming to detect deformable contours. They use a noniterative technique that searches for the optimal contour within a large neighbourhood around the initial contour. They utilize a multi-scale technique to achieve greater processing efficiency while sacrificing guaranteed optimality. 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.

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 [13, 14]. Although [13,14] 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.

2. 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 neighbours.

2.1 Local Costs

The local cost function is given by

$$l(p,q) = \omega_z \cdot f_z(q) + \omega_G \cdot f_G(q) + \omega_D \cdot f_D(p,q) + \omega_P \cdot f_P(q) + \omega_I \cdot f_I(q) + \omega_o \cdot f_o(q) \qquad (1)$$

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

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

Table 1: 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.

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 [22, 23] have dynamic cost functions. (Note that f_G is the only cost feature that has both static and dynamic components.). 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 [26, 27].

2.1.1 Laplacian Zero-Crossing (f_Z)

The primary purpose of the multi-scale Laplacian zero-crossing component, f_Z, is for edge localization [26, 28]. 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 sums 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. However, a discrete Laplacian image produces very few, if any, actual zero valued pixels. Rather, a zero-crossing is represented by two neighbouring 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 neighbour with an opposite sign; otherwise, f_Z is 1. The four horizontal/vertical neighbours of a pixel constitute the neighbourhood used to determine the zero-crossing. This creates a single pixel wide cost "canyon" and results in boundaries "snapping" to and localizing object edges. 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).

2.1.2 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

$$G = \sqrt{I_x^2 + I_y^2}. \tag{2}$$

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).

2.1.3 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 *D*(*p*) be a unit vector of the gradient direction at a point *p* and defining *D*'(*p*) as the unit vector perpendicular (rotated 90°clockwise) to *D*(*p*) (i.e., for $D(p) = [I_x(p), I_y(p)]$, $D'(p) = [I_x(p), -I_y(p)]$, then the formulation of the gradient direction feature cost is

$$f_D(p,q) = \frac{2}{3\pi}\, a\cos d_p(p,q) + a\cos d_p(p,q) \tag{4}$$

Where $d_p(p, q) = D'(p) \cdot L(p, q)$ (5)

$d_q(p, q) = \mathrm{L}(p, q) \cdot D'(p)$

are dot products and

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

Is the normalized bidirectional link or unit edge vector between pixels *p* and *q* and simply computes the direction of the link between *p* and *q* such that the difference between *p* and the direction of the link is minimized.

Links are horizontal, vertical, or diagonal (relative to the position of *q* in *p*'s neighbourhood) and point such that the dot product of *D*'(*p*) and *L*(*p*, *q*) is positive (i.e., the angle between *D*'(*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 neighbourhood link direction is to associate a high cost with an edge between two neighbouring pixels that have similar gradient directions but are perpendicular, or near perpendicular, to the link between them.

Pixel Value Features (f_P, f_I, f_O): 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 p is given by the scaling function

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

Where *I*(*p*) is the pixel value of the source image at *p*. The "inside" and "outside" pixel values [22, 23] 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 *p* is sampled a distance *k* from pin 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, $f_I(p)$, and the outside pixel value, $f_O(p)$, for a given pixel *p* is

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

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

where *D*(*p*) is the unit vector of the gradient direction, 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 p. the value can be taken as the closest pixel (default) or bilinearly interpolated from each of the four surrounding pixels.

2.1.4 Colour

Both the Laplacian zero-crossing and the gradient magnitude are computed by processing each of the three colour bands (in RGB colour 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 colour space) of the corresponding pixel. The gradient direction computation is unchanged for colour images.

2.1.5 On-the-Fly Training

Often, an object boundary may not consist of "strong" edge features. For example, Figure 1(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. As a result, when trying to track the right boundary of the 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 [25].

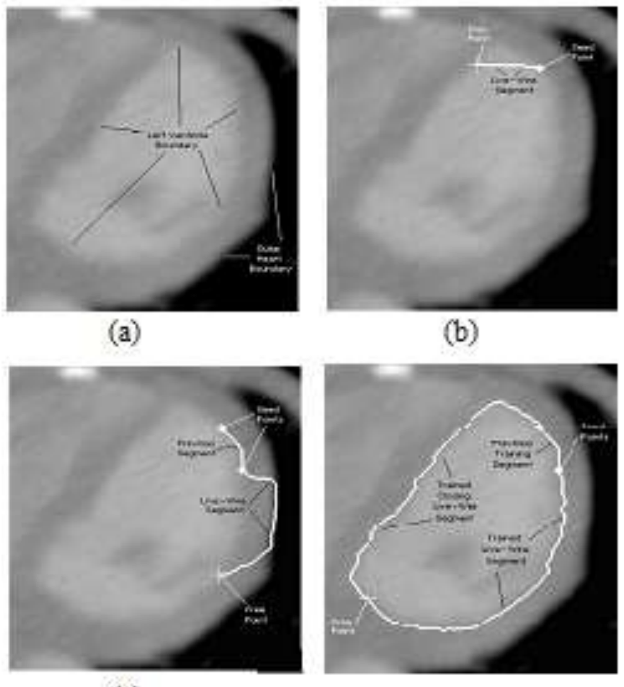

Figure 1: (a) CT scan of the heart. (b) Untrained live-wire segment (cuts across corner of left ventricle). (c) Short boundary segment and untrained live-wire segment (snaps to the stronger gradient of the outer heart wall). (d) Live-wire segments (including closing segment) trained on selected boundary segment. Notice that the short training segment is all that is needed to completely define the left ventricle boundary.

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 1(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

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

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. 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*. 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:

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

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.

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.

2.1.6 Static Neighbourhood 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 neighbourhood 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, *p*, and any neighbouring point, *q*, the static link cost map, l_S, is

$$l_S(p,q) = \lfloor M_Z \cdot f_Z(q) + 0.5 \rfloor + \lfloor M_D \cdot f_D(p,q) + 0.5 \rfloor \quad (13)$$

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 neighbours for each pixel, the precomputed static link map, l_S, requires 8N cost values for *N* image pixels.

2.1.7 Final Local Link Cost

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

$$w_N(p,q) = \begin{cases} 1; if\ L_x(p,q) \neq 0 \wedge L_y(p,q) \neq 0 \\ \frac{1}{\sqrt{2}}; if\ L_x(p,q) = 0 \vee L_y(p,q) = 0 \end{cases} \quad (14)$$

where L_x and L_y are horizontal and vertical components of the bidirectional link vector *L* defined in Eq. (6).

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 neighbour *q* of a pixel *p* is

$$\begin{aligned} l'(p,q) &= l_s(p,q) + w_N(p,q) \cdot m'_G(I_G(q)) \\ &\quad + m_P(I_P(q)) + m_I(I_I(q)) + m_O(I_O(q)) \end{aligned} \quad (15)$$

where l_s is the static link cost in Eq. (13), w_N is the neighbourhood 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. UNRESTRICTED GRAPH SEARCH

Our graph formulation is pixel based rather than crack based and we utilize a more efficient optimal graph search algorithm based on Dijkstra's [1] algorithm. The extensions to the previous optimal graph search boundary finding methods can be given in 3ways:

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

First, with the exception of [13,14,12], 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* (*p*). All other points in the image are (effectively) initialized with infinite cost. 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 neighbours. For each neighbour *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, $g_{tmp} = g(p) + l'(p, q)$. If the newly computed total cost to *q* is less than the previous cost (i.e., if $g_{tmp} < g(q)$ then $g(q)$ 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 neighbours 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 $L(i)\downarrow q$ 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 $N(p)$ be the set of pixels neighbouring p and $e(p)$ be a Boolean mapping function indicating that a point p has been expanded. Further, let ptr(q) be the optimal path pointer for the point q, then the unrestricted graph search algorithm is as follows:

Algorithm:

```
        Input:      $s$, $l(p, q)$
        Data Structures:
                $L$, $N(p)$, $e(p)$, $g(p)$
        Output:
                $ptr$
        Algorithm:
                $g(s)=0$;    $L(0)\downarrow s$
                While $L\neq\varnothing$ do begin
                        $p\leftarrow min(L)$;
                        $e(p)$=TRUE;
                for each $q\in N(p)$ such that not $e(q)$ do begin
                $g_{tmp}= g(p)+l'(p,q)$;
                if $q\in L$ and $g_{tmp}<g(q)$ then begin
                        $i=g(q)$ and $(M-1)$; $q\leftarrow L(i)$;
                end
                if $q\notin L$ then begin
                        $g(q) = g_{tmp}$;
                        $ptr(q)=p$;
                        $i=g(q)$ AND $(M-1)$;
                        $L(i)\downarrow q$;
                end
        end
end
```

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

$$l'(p,q) = l_S(p,q) + w_N(p,q)\cdot m'_G(I_G(q)) \qquad (16)$$

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 $p\leftarrow min(L)$ 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 p from L is given by

```
        $c=c-1$;
repeat
$c=c+1$;
        $i=c$ AND $(M-1)$;
        until $L(i)\neq\varnothing$
        $p\uparrow L(i)$;
```

where $p\uparrow L(i)$ denotes popping the point 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. $q\leftarrow L(i)$ denotes removing the point 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 2 demonstrates how the graph search algorithm creates a minimum cumulative cost path map (with corresponding optimal path pointers). Figure 2(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 2(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 active list (bold numbers) can change if even lower cumulative costs are computed from as yet unexpanded neighbours. This is demonstrated in Figure 2(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 2 (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 wave front grows out faster in areas of lower costs.

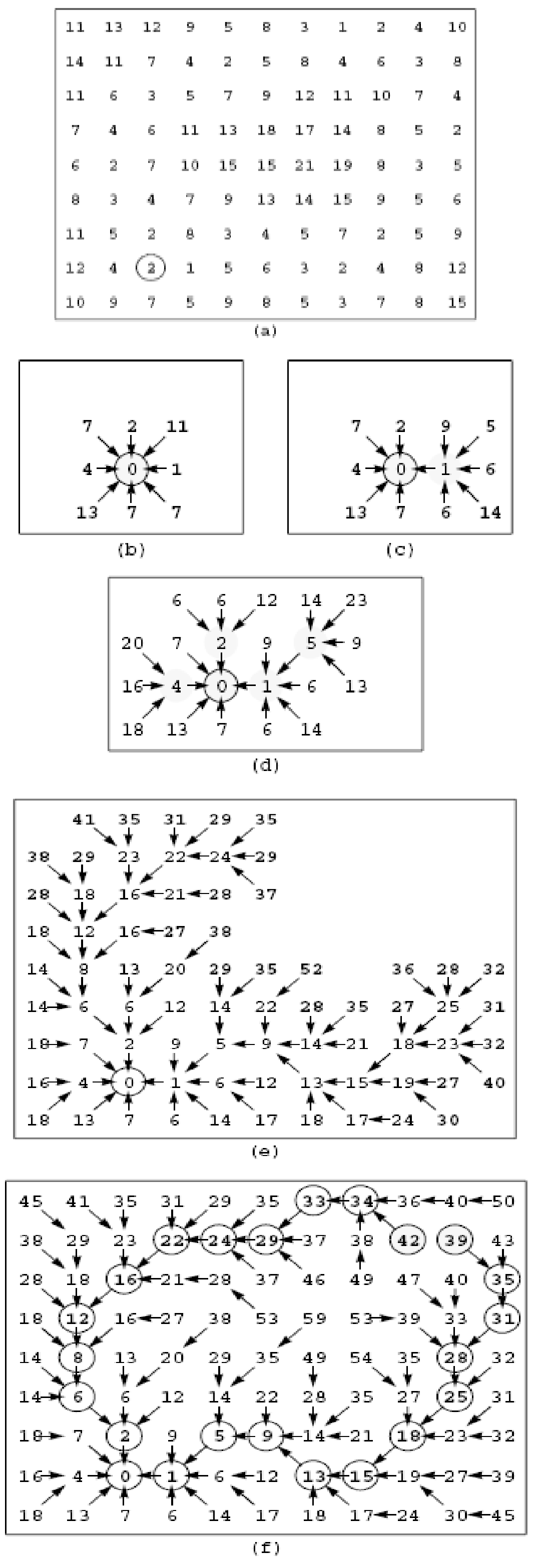

Figure 2: (a) Initial local cost matrix (b) seed point expanded (c) two points expanded (d) five points expanded (e) 47 points expanded (f) finished cumulative cost and path matrix with two of many paths.

4. EXPERIMENTAL RESULTS

Endoscopic images containing abnormal regions such as tumor growths, cancer regions are obtained from physicians. These images are subjected to segmentation using the proposed intelligence scissors method. The figures below show the experimental results obtained using the Intelligent Scissor function on the endoscopic images of the Esophagus. The abnormal region is segmented, thereby highlighting the region of disorder. The proposed method could successfully segment the region of interest.

5. CONCLUSION

This paper has presented an interactive image segmentation tool based on an unrestricted graph search. The major contributions of this work are the addition of the Laplacian zero-crossing binary feature cost that improves edge localization for optimal boundary segments. 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. 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. Cooling helps reduce the need for user input and thereby facilitates and improves the live-wire's interactivity. 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.

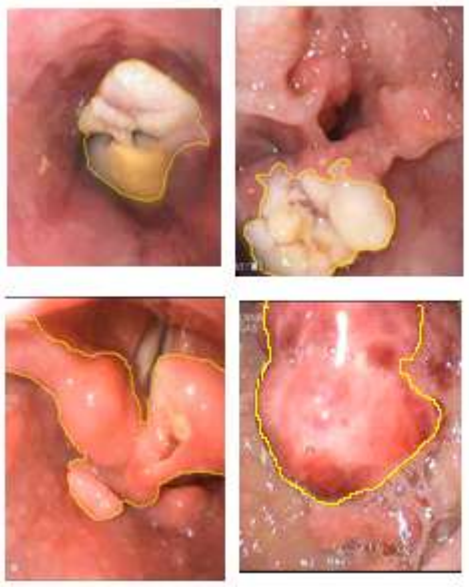

Figure 3: Showing the boundary of the segmented abnormal region of the growth in esophageal region.

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 colour and grayscale image segmentation, and line extraction from scanned document.

6. REFERENCES

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

[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] 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.

[4] 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.

[5] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

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

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

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

[14] 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.

[15] 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.

[16] 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.

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

[18] 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)

[19] 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.

[20] 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.

[21] 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

[22] 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.

[23] J. K. Udupa, S. Samarasekera, and W. A. Barrett, "Boundary Detection via Dynamic Programming," in Proceedings of the

[24] SPIE: Visualization in Biomedical Computing 92, Vol. 1808, pp. 33-39, Chapel Hill, NC, Oct. 1992.

[25] 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 Research, Vol. 13, No. 6, pp. 522-548, Dec. 1980.

[26] 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.

[27] 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.

[28] 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.