Toboggan-Based Intelligent Scissors with a Four Parameter Edge Model

Eric N. Mortensen William A. Barrett
Brigham Young University

Abstract

Intelligent Scissors is an interactive image segmentation tool that allows a user to select piece-wise globally optimal contour segments that correspond to a desired object boundary. We present a new and faster method of computing the optimal path by over-segmenting the image using tobogganing and then imposing a weighted planar graph on top of the resulting region boundaries. The resulting region-based graph is many times smaller than the pixel-based graph used previously, thus providing faster graph searches and immediate user interaction. Further, the region-based graph provides an efficient framework to compute a 4 parameter edge model, allowing subpixel localization as well as a measure of edge blur.

1. Introduction

All general purpose image segmentation techniques require some amount of human guidance due to the large variety of image sources, content and complexity. Consequently, a goal of any general image segmentation algorithm should be to accurately define the desired object boundary or region with minimal user input. A variety of segmentation tools exist ranging from user intensive lassoing to semi-automatic initialization schemes such as magic wands and active contour models. Unfortunately, lassoing or other primitive manual tracing tools are still widely used when a non-homogeneous image component must be extracted from a complex background. For this reason intelligent segmentation tools which exploit high level visual expertise but require minimal user interaction become appealing.

Recently, we presented a unique boundary based segmentation algorithm called Intelligent Scissors [12,13] which allow a human operator to interactively select an optimal boundary segment, called a "live-wire" [11,7], corresponding to a portion of the desired object edge. Intelligent Scissors allow objects to be extracted quickly and accurately using simple gesture motions with a mouse. When the mouse position comes in proximity to an object edge, a live-wire boundary "snaps" to, and wraps around the object of interest.

This paper presents improvements to our Intelligent Scissors tool based on an oversegmentation method known as tobogganing that allows for faster optimal path computation in the interactive live-wire environment. Further, the toboggan information provides an efficient framework for computing a 4 parameter edge model which provides subpixel localization, blur, and foreground/background color.

2. Previous Work

Among the boundary based segmentation strategies, active contours or snakes have received considerable attention over the last few years [1,3,8,9,18]. Active contours are initialized with an approximate boundary and then iteratively minimize the contour's energy functional to achieve an optimal boundary. The functional combines external forces such as gradient magnitude with internal forces like boundary curvature in an attempt to yield a final smooth contour that conforms to the desired object boundary. Since snakes are typically initialized with an approximate contour, the human operator is often not quite sure what the final boundary will be until the active contour settles into a minimum. Some implementations allow the user to interactively modify the energy landscape and thus nudge a snake [3,8], but still the user does not typically know exactly what the final boundary will be during interaction.

Intelligent Scissors presents a different approach to the segmentation problem. Rather than optimizing a user-initialized approximate contour, we allow the user to interactively select a boundary from a collection of optimal solutions. Thus the user knows exactly what the resulting contour will be during interaction. Intelligent Scissors achieves this goal by interactively computing the optimal path from a user selected "seed" point to all other points in the image. As the cursor moves, the optimal path from the pointer position to the seed point is displayed, allowing the user to select an optimal contour segment which visually corresponds to a portion of the desired object boundary.

3. Toboggan-Based Intelligent Scissors

Intelligent Scissors, as presented in [13], formulates the boundary detection problem as a weighted graph search where each pixel is a node and weighted edges are created between each pixel and its 8 neighbors. We continue to follow this by first partitioning the image into a collection of regions and then imposing a weighted planar graph onto the resulting region boundaries. This reduces the size of the graph which greatly speeds up the graph search and increases interactive responsiveness.

3.1. Tobogganing

Tobogganing, introduced by Fairfield [6] and extended by Yao and Hung [19], oversegments an image into small regions by sliding in the derivative terrain. The basic idea is that given the gradient magnitude of an image, each pixel determines a slide direction by finding the pixel in a neighborhood with the lowest gradient magnitude. Obviously, several pixels will represent local minima in the gradient terrain (due to either image features or noise) in that they represent the minimum gradient magnitude within their own neighborhood. Pixels that "slide" to the same local minimum are grouped together, thus segmenting the image into a collection of small regions (Fig. 1).

The regions produced by tobogganing are effectively identical to the catchment basins produced by applying the popular watershed algorithm to the gradient image[2,14,17]. However, tobogganing is much more computationally efficient than the watershed algorithm.

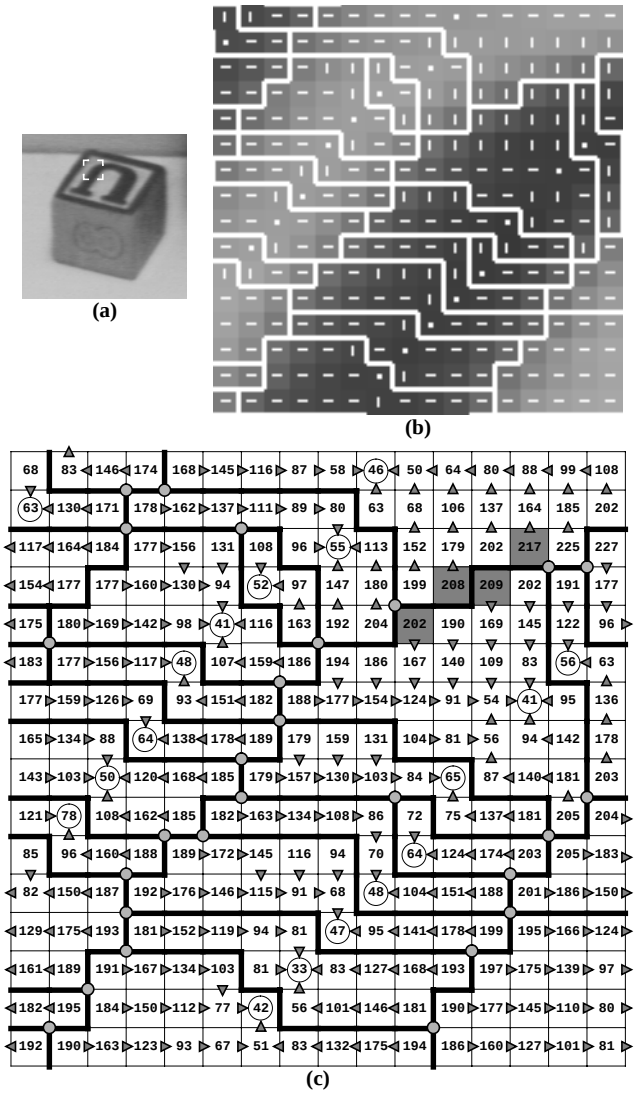


Figure 1: (a) *Original image.* ***(b)*** *Expanded section of (a) showing toboggan region boundaries, slide directions, and local minima.* ***(c)*** *Numerical illustration of (b) showing gradient magnitudes, region boundaries (thick lines), slide directions (arrow heads), and local minima (circled). Nodes (shaded circles) are created where 3 or 4 regions meet at a pixel corner while region boundary segments between two nodes become edges. The shaded pixels show an example gradient index vector (Section 3.2.2).*

3.1.1. Multi-Scale Gradient

The gradient magnitude used for tobogganing is computed using multi-scale derivatives of Gaussian kernels. If

$$N_\sigma(x,y) = \frac{1}{2\sqrt{2}\pi\sigma} e^{-\frac{1}{2}\left(\frac{x^2+y^2}{\sigma^2}\right)} \quad (1)$$

is a two-dimensional normal distribution with a standard deviation of σ, the multi-scale gradient magnitude is given by

$$G(x,y) = \sqrt{\max(G_\sigma(x,y))} \quad (2)$$

where

$$G_\sigma = \sum_b \left[I_b * \frac{\partial N_\sigma}{\partial x} \right]^2 + \left[I_b * \frac{\partial N_\sigma}{\partial y} \right]^2 \quad (3)$$

is the squared gradient magnitude summed over each color band (I_b) of the original image $\boldsymbol{I}$ as computed by convolving each I_b with a derivative of Gaussian kernel of scale σ.

3.1.2. Sliding and Grouping

Given a gradient magnitude image, we next segment the image into regions by sliding in the gradient terrain. Pixels that slide into the same local minimum are efficiently grouped into regions by assigning them a unique label. The image is scanned in row major order. If a pixel is not labelled, its four-connected neighborhood is checked to determine which neighboring pixel has the smallest gradient magnitude. The pixel then "slides" to the minimum gradient neighbor by setting a slide direction and moving to that neighbor. The process is repeated until it slides into a pixel that is already labelled or until it slides into a local minimum. If sliding reaches an unlabeled local minimum, that minimum is assigned a unique label. Now that a region label is known, unlabeled pixels encountered during sliding are labelled. Thus, the tobogganing algorithm is as follows:

Algorithm 1: Toboggan Segmentation.

```
Input:
  G(p)                    {Gradient magnitude at pixel position vector p.}
Data Structures:
  N(p)                    {4 connected neighborhood of p.}
Output:
  T(p)                    {Toboggan direction vector at p.}
  L(p)                    {Region label at p (initialized to nil for all p).}
Algorithm:
  regions←0;              {Initialize # of regions.}
  for each p do begin     {Scan image in row major order.}
    q←p;
    repeat                {Slide to labelled pixel or local minimum.}
      min←G(q); q´←q;
      for each r∈N(q) do begin    {Find lowest gradient neighbor.}
        if G(r)≤min then begin
          min←G(r); q´←r;
        end
      end
      T(q)←q´−q; q←q´;    {Set slide direction and slide.}
    until L(q)≠nil or T(q)=0
    if L(q)=nil then begin    {If local minimum is unlabeled,}
      L(q)←regions;           {  assign a unique label.}
      regions←regions+1;
    end
    r←p;
    repeat                {Repeat slide to label unlabeled pixels.}
      L(r)←L(q); r←r+T(r);
    until L(r)≠nil
  end
```

Notice that, like [19], sliding continues as long as there is a neighbor with a gradient magnitude less than or *equal to* that of the current pixel.

Note also that we use a 4-connected neighborhood as opposed to the 8-connected neighborhood of [6] and [19]. It has been our experience that using a 4-connected neighborhood, while producing more regions, isolates fine detail better than tobogganing with an 8-connected neighborhood. Further, 8-connected regions pose more topological problems when mapping region boundaries to a planar graph.

3.2. Weighted Graph

Having partitioned the image into a collection of small regions, we now convert the tobogganed regions into a weighted planar graph. The resulting graph has weighted edges and is used to compute an optimal path corresponding to a lowest cumulative cost boundary segment in the image.

3.2.1. Graph Creation

Since tobogganing produces regions that partition the image, there are no pixel-based boundaries between neighboring regions. Rather, two adjacent regions share a connected sequence of one or more "cracks" between two 4-connected pixels. As such, edges map to the boundary segment between two neighboring regions and nodes are created where three or four regions meet at a pixel corner.

Since nodes occur only at pixel corners, every node is uniquely identified by the pixel whose upper-left corner is the actual node position. Thus, nodes are specified with a pixel position vector[1], η, and edges are represented as an ordered quadruple, (η_{src}, η_{dst}, l_l, l_r), indicating the source and destination nodes as well as the region labels on its left and right side, respectively.

The graph is created by tracking the boundary of each region using an "over-the-shoulder" contour following algorithm. Given the definitions of $L(\boldsymbol{p})$ and $\boldsymbol{N}(\boldsymbol{p})$ in Algorithm 1 and pixels $\boldsymbol{p}$ and $\boldsymbol{q}$ such that $\boldsymbol{q} \in \boldsymbol{N}(\boldsymbol{p})$ and $L(\boldsymbol{p}) \neq L(\boldsymbol{q})$ — i.e., neighboring pixels $\boldsymbol{p}$ and $\boldsymbol{q}$ span the boundary between adjacent regions—then the crack between $\boldsymbol{p}$ and $\boldsymbol{q}$ is defined as the ordered pair ($\boldsymbol{p}$, $\boldsymbol{q}$) and

$$\boldsymbol{d}_{\boldsymbol{p},\boldsymbol{q}} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} (\boldsymbol{p} - \boldsymbol{q}) \tag{4}$$

is the crack direction vector pointing clockwise relative to $\boldsymbol{p}$. Thus, if $l = L(\boldsymbol{p})$, $\boldsymbol{q_d} = \boldsymbol{q} + \boldsymbol{d}_{\boldsymbol{p},\boldsymbol{q}}$ and $\boldsymbol{p_d} = \boldsymbol{p} + \boldsymbol{d}_{\boldsymbol{p},\boldsymbol{q}}$ then

$$next(p, q) = \begin{cases} (\boldsymbol{q_d}, \boldsymbol{q}); & \text{if } L(\boldsymbol{q_d}) = l \\ (\boldsymbol{p_d}, \boldsymbol{q_d}); & \text{if } L(\boldsymbol{q_d}) \neq l \text{ and } L(\boldsymbol{p_d}) = l \\ (\boldsymbol{p}, \boldsymbol{p_d}); & \text{otherwise} \end{cases} \tag{5}$$

is the next clockwise pixel crack on the boundary of region l relative to the crack ($\boldsymbol{p}$, $\boldsymbol{q}$).

Definition: $\boldsymbol{B}(l)$ is the ordered n-tuple, ($(\boldsymbol{p}_1,\boldsymbol{q}_1)$, $(\boldsymbol{p}_2,\boldsymbol{q}_2)$, …, $(\boldsymbol{p}_n,\boldsymbol{q}_n)$), of pixel cracks that represent, in clockwise order, the boundary of a 4-connected region with label l such that all of the following properties hold:

1. $L(\boldsymbol{p}_i) = l$ for all $1 \le i \le n$
2. $L(\boldsymbol{q}_i) \neq l$ for all $1 \le i \le n$
3. $(\boldsymbol{p}_{i+1}, \boldsymbol{q}_{i+1}) = next(\boldsymbol{p}_i, \boldsymbol{q}_i)$ for all $1 \le i < n$
4. $(\boldsymbol{p}_1, \boldsymbol{q}_1) = next(\boldsymbol{p}_n, \boldsymbol{q}_n)$
5. $L(\boldsymbol{q}_n) = L(\boldsymbol{q}_1)$ iff $L(\boldsymbol{q}_i) = L(\boldsymbol{q}_j)$ for all $1 \le i, j \le n$

Processing each region, l, we note that there is a node on the boundary whenever $L(\boldsymbol{q}_i) \neq L(\boldsymbol{q}_{i+1})$, corresponding to the node position

$$\eta_i = \frac{1}{2}\left(\boldsymbol{p}_i + \boldsymbol{q}_i + \boldsymbol{d}_{\boldsymbol{p}_i,\boldsymbol{q}_i} + \begin{bmatrix} 1 \\ 1 \end{bmatrix}\right) \tag{6}$$

Further, if $L(\boldsymbol{q}_j) \neq L(\boldsymbol{q}_{j+1})$ for $j > i$ such that $L(\boldsymbol{q}_k) = L(\boldsymbol{q}_{i+1})$ for $i < k \le j$ then there is an edge, $\boldsymbol{e} = (\eta_i, \eta_j, L(\boldsymbol{q}_j), l)$, defined between nodes η_i and η_j that separates regions $L(\boldsymbol{q}_j)$ and l where $\boldsymbol{e}_{crack} = ((\boldsymbol{p}_{i+1},\boldsymbol{q}_{i+1}), (\boldsymbol{p}_{i+2},\boldsymbol{q}_{i+2}), \ldots, (\boldsymbol{p}_j,\boldsymbol{q}_j))$ is the ordered sequence of pixel cracks defining the edge.

1. We use η to represent a position vector corresponding to a node in order to avoid confusion with a regular position vector such as $\boldsymbol{p}$ or $\boldsymbol{q}$.

Fig. 1(c) illustrates how the tobogganed regions form a planar graph. Graph nodes are indicated with shaded circles, ●, while edges, shown as thick lines, correspond to the portion of a region boundary connecting two nodes.

3.2.2. Edge Weights

Since region boundaries already localize object edges, there is no need for the Laplacian zero-crossing cost used in [13]. Rather, edge costs are currently computed using only the gradient magnitude in a two step process. The first step creates a gradient index vector for each edge and the second step computes an edge's gradient cost by remapping and summing the index vector. The gradient index, given by

$$idx_G = \left\lfloor (n_G - 1)\frac{G}{\max(G)} + \frac{1}{2} \right\rfloor \tag{7}$$

scales the gradient magnitude to integer values between 0 and n_G-1 (which is the domain of the remapping function used in the second step).

Given an edge $\boldsymbol{e}$ such that $\boldsymbol{e}_{crack} = ((\boldsymbol{p}_1,\boldsymbol{q}_1), (\boldsymbol{p}_2,\boldsymbol{q}_2), \ldots, (\boldsymbol{p}_n,\boldsymbol{q}_n))$, the gradient index vector of $\boldsymbol{e}$, $\boldsymbol{idx}_{G,\boldsymbol{e}} = (idx_G(\boldsymbol{r}_1), idx_G(\boldsymbol{r}_2), \ldots, idx_G(\boldsymbol{r}_m))$, is an ordered m-tuple of gradient indices where $m \le n$ and $(\boldsymbol{r}_1, \boldsymbol{r}_2, \ldots, \boldsymbol{r}_m)$ is the 8-connected pixel sequence defined algorithmically as follows:

```
r_prv←(-1,-1); j←1;                          {Init. previous pixel position & j.}
for i = 1 to n do begin
  if idx_G(p_i) > idx_G(q_i) then r_tmp←p_i;  {Set r_tmp to crack pixel}
  else                       r_tmp←q_i;       { with largest gradient.}
  if r_tmp ≠ r_prv then begin                 {If new pixel position, }
    r_j←r_tmp; j←j+1; r_prv←r_tmp;            { add it to path.}
  end
end
m←j-1;                                        {Set path length.}
```

Fig 1(c) highlights, in the upper-left, the gradient index vector for an example edge. Notice that even though the edge is composed of 5 pixel cracks, its gradient index vector is only 4 gradient values (since the maximum gradient positions for two of the cracks are identical). Consequently, the gradient index vector for diagonal edges is typically shorter than the crack count, thus providing a mechanism to scale the edge cost based on estimated Euclidean length.

As mentioned, an edge's final cost function is computed by remapping and summing its gradient index vector. The remapping function is initially defined as

$$map_G(i) = \left\lfloor \left(1 - \frac{i}{n_G - 1}\right)M + \frac{1}{2} \right\rfloor \tag{8}$$

which is an inverse linear ramp ranging from M down to 0 where M is the maximum cost per boundary unit. map_G is implemented as a lookup table in order to adapt dynamically and thereby accommodate on-the-fly training (as described in [13]).

The final edge cost function is given by

$$f(e) = \sum_{j=1}^{m} map_g(idx_G(\boldsymbol{r}_j)) \tag{9}$$

and is simply the summation of the edge's remapped gradient index vector.

3.3. Optimal Graph Search

Computation of optimal paths corresponding to minimum cost contour segments is accomplished using Dijkstra's [4] optimal graph search in much that same manner as in [13]. However, graph edges in [13] had unit length costs, thereby bounding the cost range on the "active" list and allowing for an efficient bin sort algorithm. Since graph edges in this paper can be several units long, the active list now employs an efficient two-level bin sort algorithm to allow for a much wider cost range.

Given a user selected "seed" node, η_s, the optimal graph search algorithm is as follows:

Algorithm 2: Optimal Graph Search.

```
Input:
  η_s                 {Seed node.}
Data Structures:
  L                   {List of active nodes sorted by total cost (initially empty).}
  edge(η)             {Edge set containing edges emanating from η.}
  done(η)             {Boolean function indicating if η has been expanded.}
  g(η)                {Total cost function from η_s to η.}
Output:
  opt(η)              {Edge from η indicating optimal path back to η_s.}
Algorithm:
  g(η_s)←0;  L←η_s;        {Initialize active list with zero cost seed node.}
  while L ≠∅ do begin      {While still nodes to expand:}
    η←min(L);              {Remove minimum cost node η from active list.}
    done(η)←TRUE;          {Mark η as expanded (i.e., processed).}
    for each e ∈ edge(η) do begin
      η_n←e_dst(η);                    {Get neighbor connected to edge.}
      if not done(η_n) then begin
        g'←g(η)+f(e);                  {Compute total cost to neighbor.}
        if η_n ∈ L and g' < g(η_n) then  {Remove higher cost }
          η_n←L;                       {  neighbor from list.}
        if η_n ∉ L then begin          {If neighbor not on list, }
          g(η_n)←g'; opt(η_n)←e;       {  assign total cost, set opt. edge}
          L←η_n;                       {  and place on (or return to) }
        end                            {  active list.}
      end
    end
  end
```

While the optimal graph search algorithm presented here is similar to our previous work in [13], the reduced graph size allows this technique to compute optimal paths anywhere from two to 10 or more times faster, depending on the average size of the tobogganed regions, despite the fact that we now use a more complex two-level bin sort.

3.4. Four Parameter Edge Model

One advantage of using toboggan region boundaries as a basis for the Intelligent Scissors graph search is the ability for subpixel localization of an object edge by applying an edge model to the region boundaries. As with [5], we assume that an edge is modeled as a Gaussian blurred step edge. If α is the step amplitude and β is the base (pedestal offset), then the preferred edge model is given by

$$\frac{\alpha}{2}\left(erf\left(\frac{x - x_0}{\sqrt{2}\sigma}\right) + 1\right) + \beta \tag{10}$$

where σ is the standard deviation of the Gaussian blur and x_0 is the step edge location. Unlike [5], we model edges by extracting profile data for each edge crack and fitting the data to a model using a modified 4 parameter Marquardt minimization [15,16]. However, because of the complexity of the partials of Eq. (10), especially when embedded in a summed squared residuals function (i.e., a χ^2 function), we approximate Eq. (10) with a sigmoid—since its partials are much more tractable and it very closely approximates the error function. Thus, the edge model we use is

$$\frac{\alpha}{1 + e^{(x_0 - x)/s}} + \beta \tag{11}$$

where s is the spread or blur (similar to σ in Eq. (10)).

Note that Eq. (11) only models the transition between two gray levels (or colors). As such, we would like to avoid including transition information from edges other than the one being modeled. Since tobogganing tends to slide away from edges and stops sliding before climbing another edge, the slide information provides a reasonable mechanism for extracting transition data.

We extract the edge profile following the slide path from both sides of an edge crack and projecting the position and color data onto a domain and range vector respectively. Given an edge crack $(\boldsymbol{p},\boldsymbol{q})$ we define a slide path for $\boldsymbol{p}$ as

$$\boldsymbol{slide}(\boldsymbol{p}) = (\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_n) \tag{12}$$

where $\boldsymbol{p}_1 = \boldsymbol{p}$, $\boldsymbol{p}_{j+1} = \boldsymbol{p}_j + \boldsymbol{T}(\boldsymbol{p}_j)$ for $1 \le j < n$, $\boldsymbol{p}_n$ is the local minima for the tobogganed region, and $\boldsymbol{T}(\boldsymbol{p}_j)$ is the toboggan slide direction defined in Algorithm 1. The slide path for $\boldsymbol{q}$ is defined in like manner and the two slide paths are concatenated into a profile sample vector

$$\begin{aligned}\boldsymbol{sample}(\boldsymbol{p},\boldsymbol{q}) &= (\boldsymbol{s}_1, \boldsymbol{s}_2, \ldots, \boldsymbol{s}_{n+m}) \\ &= (\boldsymbol{p}_n, \ldots, \boldsymbol{p}_2, \boldsymbol{p}_1, \boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_m)\end{aligned} \tag{13}$$

by reversing $\boldsymbol{slide}(\boldsymbol{p})$ and appending $\boldsymbol{slide}(\boldsymbol{q})$.

We define the domain projection vector, $\boldsymbol{v}_x$, with origin, $\boldsymbol{o}_x$, as the normalized gradient direction vector originating at the crack $(\boldsymbol{p},\boldsymbol{q})$—the normalized sum of the gradient direction vectors at $\boldsymbol{p}$ and $\boldsymbol{q}$. Each domain value, x_i, is computed by projecting the corresponding relative sample vector, $\boldsymbol{s}_i - \boldsymbol{o}_x$, onto $\boldsymbol{v}_x$. Thus, $x_i = (\boldsymbol{s}_i - \boldsymbol{o}_x) \cdot \boldsymbol{v}_x$. Figure 2 illustrates the edge model computation for a sample edge crack.

For grayscale images, the obvious y_i's are simply the pixel gray-levels for each corresponding x_i (see Fig. 2(b)). However, no such mapping is obvious for color images. We thus define the range vector, $\boldsymbol{v}_y = \boldsymbol{I}(\boldsymbol{q}_m) - \boldsymbol{o}_y$, where $\boldsymbol{I}(\boldsymbol{q}_m)$ is the image color vector at the pixel position $\boldsymbol{q}_m$ and $\boldsymbol{o}_y = \boldsymbol{I}(\boldsymbol{p}_n)$. Each range value, y_i, is then computed in a similar fashion as the x_i's. Specifically, $y_i = (\boldsymbol{I}(\boldsymbol{s}_i) - \boldsymbol{o}_y) \cdot \boldsymbol{v}_y / \|\boldsymbol{v}_y\|$.

As mentioned, we fit the edge model given in Eq. (11) to the profile data using a Marquardt minimization. One advantage of the Marquardt method is that upon reaching an acceptable minimum, the curvature matrix can be recalculated and used to estimate the covariance matrix of standard errors in the fitted parameters. The standard errors can be used to evaluate the "reliability" of each parameter. For example, we use $\sigma(x_0)$—the estimated standard deviation in the fit of x_0—to smooth the displayed object path, as noted below.

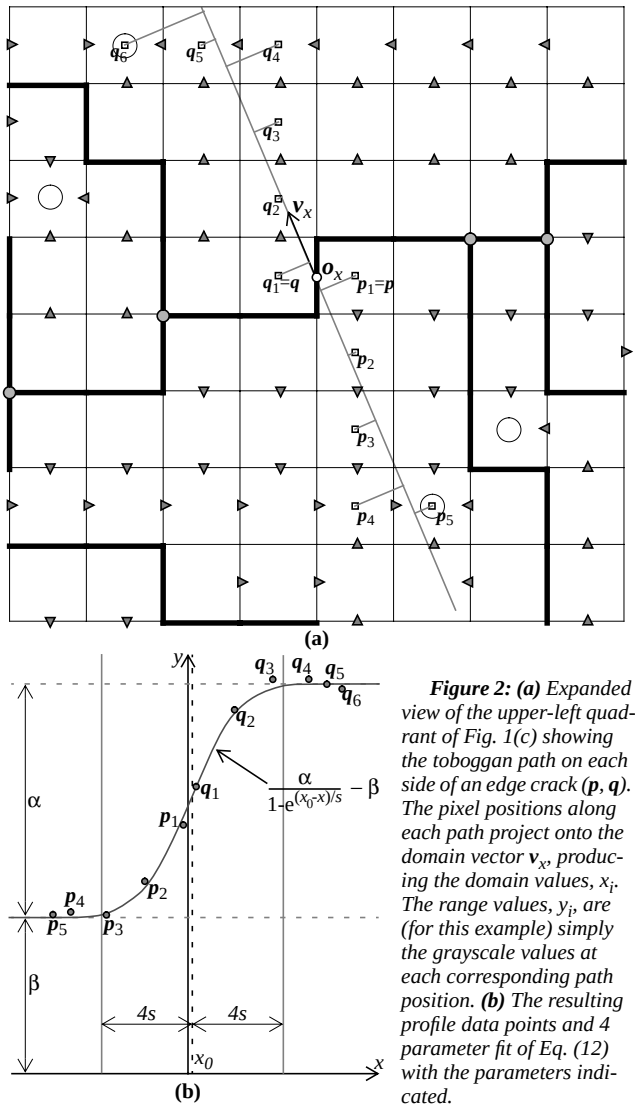


Figure 2: (a) *Expanded view of the upper-left quadrant of Fig. 1(c) showing the toboggan path on each side of an edge crack (**p**, **q**). The pixel positions along each path project onto the domain vector $\mathbf{v}_x$, producing the domain values, x_i. The range values, y_i, are (for this example) simply the grayscale values at each corresponding path position.* ***(b)*** *The resulting profile data points and 4 parameter fit of Eq. (12) with the parameters indicated.*

3.5. Live-Wire Boundary Selection

Once the optimal path to every node is computed, the live-wire allows interactive selection of the optimal path corresponding to a segment of the desired object boundary. As a pointing device moves, the cursor position is used to select a node and display the optimal path from that node back to the seed node. The optimal path is displayed as a polyline fit to the subpixel midpoints of the path edges. The midpoint for each edge crack is defined as $\mathbf{o}_x + x_0\mathbf{v}_x$ where $\mathbf{o}_x$ and $\mathbf{v}_x$ are the domain origin and vector (as defined previously) and x_0 is the subpixel edge position given in Eq. (11). The displayed polyline is fit to the sequence of midpoints by incrementally determining the maximum k such that a line between midpoints $\mathbf{p}_i$ and $\mathbf{p}_{i+k}$ is at most c standard deviations from all intermediate midpoints. That is, having determined the polyline fit up to $\mathbf{p}_i$, the next line segment finds the maximum k such that for all midpoints $\mathbf{p}_j$, $i < j < i+k$, the line connecting $\mathbf{p}_i$ and $\mathbf{p}_{i+k}$ is at most a distance of $c \cdot \sigma(x_{0,j})$ away from each $\mathbf{p}_j$ where $x_{0,j}$ is the subpixel edge position for $\mathbf{p}_j$. The smoothing constant, c, can be interactively adjusted to allow a tighter or looser fit of the polyline.

Due to the underlying optimal nature of the displayed live-wire path, as the cursor position moves away from the seed node in proximity to an object edge, the live-wire snaps to the object edge. When further movement of the cursor causes the live-wire path to depart from the desired object boundary, placement of a new seed node prior to the point of departure reinitiates the optimal graph search. This allows the user to continue defining the object boundary, thereby creating a piece-wise optimal boundary.

Since it is impossible to exactly specify a node located on the interpixel grid using a pixel-based pointing device (and since it would be impractical even if it were possible), edge snapping is provided to select the edge that is "nearest" to the current pointer position [10,12,13]. The nearest edge is determined by computing the weighted Euclidean distance to each edge around the region containing the cursor. An edge's weight is proportional to the average of the remapped gradient index vector. The edge "nearest" to the cursor is selected and used to select the nearest node.

Since there is only a single optimal path from any given node back to a seed node, pixel-based Intelligent Scissors requires a minimum of two seed points to define a closed boundary. However, the region-based version has the capability of selecting an edge and then displaying the path back to a seed node from both nodes connected by that edge. If the two optimal paths from each node do not overlap, then the edge and the two optimal paths define a closed boundary. Consequently, in some cases a single seed node is all that is needed to define the desired closed boundary. For example, Figure 3(a) shows the block from Fig. 1(a) and the resulting closed boundary defined with a single seed point. If an image is free of noise, then it is possible to have a single graph edge that completely encloses an object. In such cases, the object boundary can be selected without any seed nodes. This is demonstrated in Fig. 3(b) where all of the boundaries in this synthetic image are specified without any seed points since each boundary is completely defined with a single graph edge. The edge that is "nearest" to the cursor position is displayed even when there is no optimal path information. Thus, any given boundary in Fig. 3(b) can be defined by moving the cursor closer to it than any other boundary and specifying that the boundary is complete.

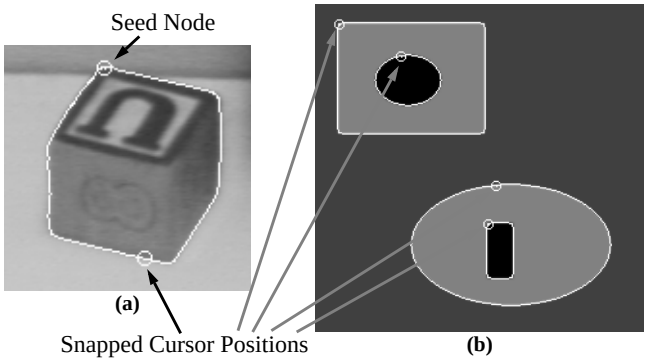


Figure 3: (a) *Live-wire boundary (in white) defined with only a single seed point.* ***(b)*** *Boundaries defined without any seed points since each boundary corresponds to a single graph edge.*

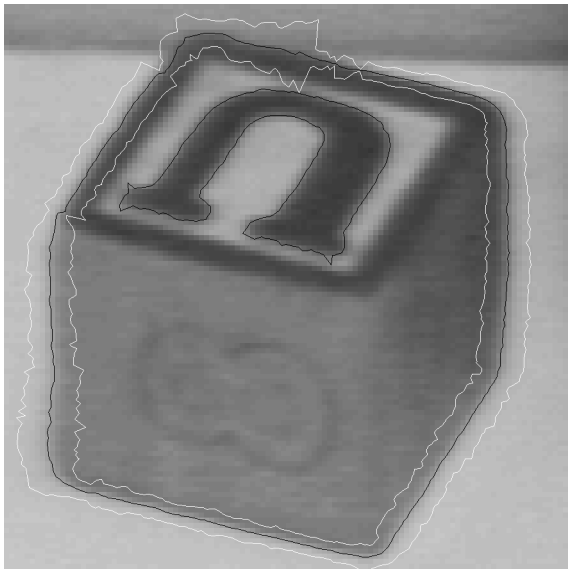


Figure 4: *Block in Fig. 1(a) pixel replicated by a factor of 8 to show the zero tolerance subpixel localization of the block and 'U' boundaries (black) as well as the transition cutoff (white) of the block boundary.*

4. Results and Discussion

We have not yet performed any comprehensive quantitative analysis comparing the accuracy, reproducibility, and boundary definition times between toboggan-based Intelligent Scissors and that in [13]. However, as the authors of [13], we have extensive experience and familiarity with both techniques and submit the following observations:

- The seed point boundary definition range (i.e., the amount of an object boundary defined with a single seed point) appears to be nearly identical for both techniques. Hence, both methods typically require close to the same number of seed points (though in some cases, this technique has the advantage of edge selection to define a boundary with only a single seed point, or in some cases no seed point).
- In general, boundary definition times depend on the time required to manually position seed points. Since the region-based graph search allows for faster optimal path computation, this version offers better interactive responsiveness within the live-wire environment, thus permitting quicker mouse movements and faster seed point positioning. Further, the improved efficiency of this technique permits other computations during interaction, such as computing edge models and fitting midpoints with a polyline.
- Reproducibility seems to be at least as good for this technique since errors in reproducibility occur mostly in the vicinity of seed points.
- We feel, due to the subpixel boundary positioning and based on visual results, such as those shown here (i.e., Figures 4, 5(b), and 6(b)), that this technique appears to be more accurate than [13].

Figure 4 shows the estimated subpixel boundary locations for the block of Fig. 1(a) and the "U" on top of the block. Also shown are the transition cutoff on either side of the block boundary. The cutoff is set at 4*s* where *s* is the spread or blur parameter in Eq. (11). These boundaries are drawn with a zero error tolerance, meaning that the smoothing factor *c* is set to zero and no smoothing occurs. Of note is that the boundaries exhibit reasonable consistency even though the edge models are computed independently for each boundary element. For example, the transition cutoff lines indicate that the horizontal point spread of the imaging device is larger than the vertical point spread. This is in fact the case with the video camera used to acquire this image. Also note that there are a couple of spots, such as on the serifs of the 'U' or near the top corner of the block, where, due to the proximity of another edge, there was insufficient or unreliable data points for the parameter fit to reliably compute x_0 and/or *s*. However, since these values are not reliable, they will be smoothed out when drawn with a nonzero error tolerance (such as $c = 1$).

Figures 5 and 6 illustrate some simple object boundaries defined within a couple of seconds[2] and requiring only one or two seed nodes while Fig. 7 shows a more complex boundary requiring several seed nodes. In general, the boundaries correspond well with the desired object edge.

Preprocessing requires approximately 10 seconds for a 512x512 grayscale image on a 99 MHz HP 735 workstation and involves computing the image gradient using a multiscale kernel, tobogganing in the gradient terrain to define the regions, and transforming the region boundaries into a weighted graph. The edge model computation is performed during live-wire interaction on an "as need" basis. Also, since we only store cost information for the edges of a region based graph instead of every pixel, the memory requirements are smaller than that of [13].

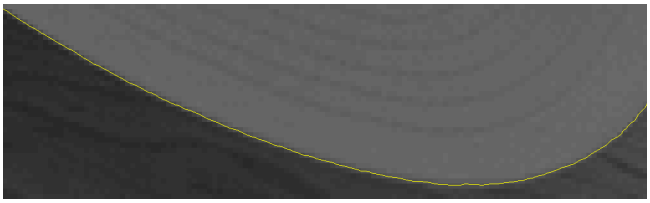
Finally, the region-based graph provides a framework for a variety of future refinements that would be difficult to achieve with the pixel-based tool. Some of the improvements include incorporating reliable region information into the cost function such as foreground/background color and multiple path exclusion with contour untangling to facilitate definition of long, thin image features (such as hair, sticks, etc.) by allowing the live-wire to snap to the strong side of such objects only once. We are in the process of adding these refinements and believe they will improve the effectiveness of Intelligent Scissors even more.

5. Conclusions

Compared to the original Intelligent Scissors tool, toboggan-based Intelligent Scissors reduces computational requirements during the critical interactive optimal boundary selection process. Further, the region-based graph provides a framework to compute an edge model, allowing for subpixel boundary localization and edge blur. In addition to the improvements mentioned previously, other extensions could include hierarchical graph construction with dynamic detail adjustment for improved computational performance and texture based cost functions.

2.Times stated are for the boundaries shown and are based on the best time achieved by an experienced user measured from the time the first seed node is placed to completion of boundary definition.

(a)



(b)

Figure 5: *Grayscale image of hat. Size: 256x256.* ***(a)*** *Boundary definition: 1.4 seconds with 1 seed node. (b) Magnified section of hat showing zero tolerance (i.e., no smoothing) subpixel boundary localization.*

(a)

(b)

Figure 6: *Full color image of a tulip. Size: 256x256.* ***(a)*** *Boundary definition: 1.6 seconds with 2 seed nodes. (b) Magnified section of tulip showing zero tolerance subpixel boundary localization.*

6. References

[1] A. A. Amini, T. E. Weymouth, and R. C. Jain, "Using Dynamic Programming for Solving Variational Problems in Vision," *IEEE PAMI*, **12**(2): 855-866, Sept. 1990.

[2] S. Beucher and C. Lantuéjoul, "Use of Watersheds in Contour Detection," in *Proc. Int. Workshop on Image Processing, Real-Time Edge and Motion Detection/Estimation*, Sept. 1979.

[3] D. Daneels, et al., "Interactive Outlining: An Improved Approach Using Active Contours," in *SPIE Proc. Storage and Retrieval for Image and Video Databases*, **1908**: 226-233, Feb. 1993.

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

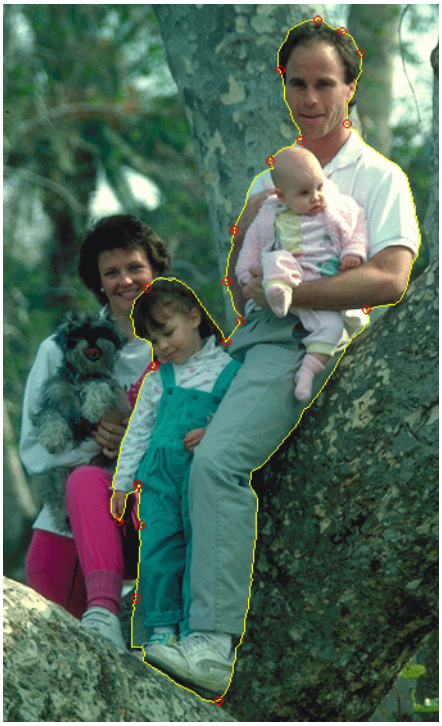


Figure 7: *Full color image of a family. Size: 340x560. Boundary definition: 18.5 seconds with several seed nodes.*

[5] J. H. Elder and S. W. Zucker, "Scale Space Localization, Blur, and Contour-Based Image Coding," in *CVPR*, 27-34, June, 1996.

[6] J. Fairfield, "Toboggan Contrast Enhancement for Contrast Segmentation," in *IEEE Proc. 10th Int. Conf. on Pattern Recog.*, **1**: 712-716, 1990.

[7] A. X. Falcão, et al., "User-Steered Image Segmentation Paradigms: Live Wire and Live Lane," *GMIP*, **60**(4): 233-260, July 1998.

[8] M. Kass, A. Witkin, and D. Terzopoulos, "Snakes: Active Contour Models," *Int. Journal of Computer Vision*, **1**(4): 321-331, Jan. 1988.

[9] D. Geiger, A. Gupta, L. A. Costa, and J. Vlontzos, "Dynamic Programming for Detecting, Tracking, and Matching Deformable Contours," *IEEE PAMI*, **17**(3): 294-302, Mar. 1995 (Correction in *PAMI*, **18**(5): 575, May 1996)

[10] M. Gleicher, "Image Snapping," in *Proc. ACM SIGGRAPH 95: Computer Graphics and Interactive Techniques*, 183-190, Aug. 1995.

[11] E. N. Mortensen, et al., "Adaptive Boundary Detection Using 'Live-Wire' Two-Dimensional Dynamic Programming," in *IEEE Proc. Computers in Cardiology*, 635-638, Oct. 1992.

[12] E. N. Mortensen and W. A. Barrett, "Intelligent Scissors for Image Composition," in *Proc. of the ACM SIGGRAPH 95: Computer Graphics and Interactive Techniques*, pp. 191-198, Aug. 1995.

[13] E. N. Mortensen and W. A. Barrett, "Interactive Segmentation with Intelligent Scissors," *GMIP*, **60**(5): 349-384, Sept. 1998.

[14] L. Najman and M. Schmitt, "Geodesic Saliency of Watershed Contours and Hierarchical Segmentation," *IEEE PAMI*, **18**(12): 1163-1773, Dec. 1996.

[15] J. C. Nash, *Compact Numerical Methods for Computers*, ch. 17: 207-217, Adam Hilger, 1990.

[16] W. H. Press, et al., *Numerical Recipes in C*, ch. 15: 681-688, Cambridge University Press, 2nd ed., 1992.

[17] L. Vincent and P. Soille, "Watersheds in Digital Spaces: An Efficient Algorithm Based on Immersion Simulations," *IEEE PAMI*, **13**(6): 583-598, June 1991.

[18] D. J. Williams and M. Shah, "A Fast Algorithm for Active Contours and Curvature Estimation," *CVGIP: Image Understanding*, **55**(1): 14-26, Jan. 1992.

[19] X. Yao and Y. P. Hung, "Fast Image Segmentation by Sliding in the Derivative Terrain," in *SPIE Proc. Intelligent Robots and Computer Vision X: Algorithms and Techniques*, **1607**: 369-379, Nov. 1991.