Contour Tracking based on Intelligent Scissors and Snakes

(Visual Modeling Course Project)

Chenfanfu Jiang
Computer Science Department
University of California, Los Angeles
Email:cffjiang@cs.ucla.edu

Garett Ridge
Computer Science Department
University of California, Los Angeles
Email: garett.ridge@gmail.com

Jingyi Fang
Computer Science Department
University of California, Los Angeles
Email: kingpotter1990@gmail.com

***Abstract*—A small program is presented that compares multiple techniques for image segmentation and video contour tracking. The techniques used here are Intelligent Scissors (live wire), Snakes (active contours) using the greedy method, and Snakes using the dynamic programming method. Experimental results show the advantages and pitfalls of each technique. Once combined, the techniques available to this tool supplement each other in order to successfully uncover visual contours that are less than conspicuous. We conclude by suggesting a potential application of our tool towards ongoing astrophysics research.**

Contents

I. Introduction

Image segmentation and video tracking are among the oldest and most heavily studied problems in computer vision research. The basic goal is to find the boundary of an object in a 2D image and to represent it as a curve.

Active contours are a popular class of methods for segmentation. A curve is initialized in an arbitrary or user-defined position, whereupon it moves toward the solution boundary as it iterates. Imaginary forces from the image data or from user interaction attract the curve towards its destination, thus earning the "active" in the method's name. Snakes[2], Intelligent Scissors[5] and Level Sets are three examples of active contours. The first is based on energy minimization, the second on shortest paths through graphs, and third upon evolving a curve as the zeroset of some particular function.

The original snake model is based on minimization of an energy functional, which is the sum of an internal energy (due to stretching and bending), an image energy (based on where the snake lies in the image), and an external energy (constraints and user interaction).

$$\begin{aligned} E^*_{snake} &= \int_0^1 E_{snake}(\mathbf{v}(s))ds \\ &= \int_0^1 E_{int}(\mathbf{v}(s)) + E_{image}(\mathbf{v}(s)) + E_{con}(\mathbf{v}(s))ds. \end{aligned} \tag{1}$$

Here the internal energy is defined as

$$E_{int}(\mathbf{v}(s)) = \frac{1}{2}(\alpha(s)\left|\mathbf{v}_s(s)\right|^2 + \beta(s)\left|\mathbf{v}_{ss}(s)\right|^2), \tag{2}$$

where $\alpha(s)$ and $\beta(s)$ are weights for continuity and curvature, respectively. The first term is coined the "membrane term" of the snake since it causes the snake to behave like a rubber band trying to pull its points closer together; the second can be imagined as a "thin plate" term since it causes the snake to try to straighten itself out, as would a thin elastic sheet of metal extending outward from the image (or simply a metal rod).

The variational problem is then discretized and solved iteratively. Here we will not go further into how this is solved with finite differences.

Since the original snake, researchers have developed several different forms and implementations for the same model[3][4][1]. In this paper, we will implement the Greedy Algorithm Snake[1] and the Dynamic Programming Snake[4], both of which are simple to implement and modify. We will also implement Intelligent Scissors[5]. Notably, the Snake sometimes performs worse than Intelligent

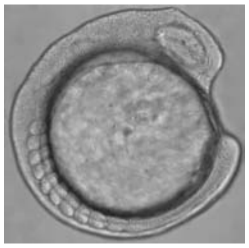

Figure 1. Zebra Fish Embryo

Scissors, so we combine the two techniques to achieve better video contour tracking.

II. Intelligent Scissors

A. Method Introduction

By treating an image as a weighted directed graph, the problem of image segmentation is transformed into the problem of searching for a minimum cost path in the graph. Eight neighboring edges enter and leave each pixel; all of them have weights, which are determined by local cost functions. Two local cost functions were combined linearly by a user input weight ω:

$$f(\mathbf{p}) = \omega * f_{edge} + (1-\omega) f_{gradient}; \tag{3}$$

The edge that ends in pixel $\mathbf{p}$ will have weight $f(\mathbf{p})$. Further descriptions of the local cost functions follow:

- f_{edge}: OpenCV is used to detect Canny edges in the image and set a binary cost. Pixels that have 0 cost if they lie on an edge, or 1 if they do not.
- $f_{gradient}$: Gradient amplitude is calculated from the partial gradient vector (I_x, I_y) of the image: $G = \sqrt{I_x^2 + I_y^2}$. To impose a higher penalty for lower gradient, $f_{gradient}$ is defined as:

$$f_{gradient} = \frac{max(G) - G}{max(G)} = 1 - \frac{G}{max(G)} \tag{4}$$

After the local cost function is determined, the image is transformed into a graph. Dijkstra's famous greedy algorithm[7] is used for global minimum cost path finding.

B. Our Implementation

There are two novelties within our implementation:

- Instead of using the Laplacian Zero-Crossing as an edge's local cost, we used the Canny edge detection algorithm for a binary edge cost definition, believing that the Canny edge algorithm is better suited for for edge detection.
- We added automatic contour closure detection; our program automatically detects whether the user selected contour is closed or not, and once it is closed, it runs and displays a "Greedy Snake" as it tracks the specified contours in the video.

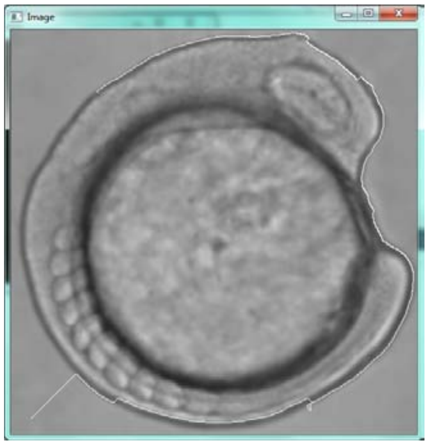

Figure 2. Minimum Cost Path

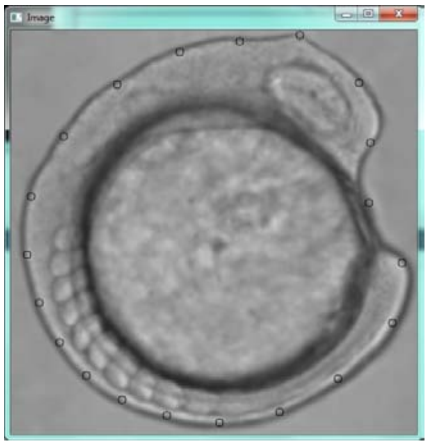

Figure 3. Auto Closed Contour Detection

C. Usage and Results

Figure (1), showing a zebrafish embryo, is the original image for segmentation. Figure (2) shows the white path between the seed and the cursor as determined by Dijkstra's algorithm. Figure (3) shows the program as it samples the contour after having detected its closure, obtaining the points that will later initialize a "Greedy Snake". Figure (4) shows the Canny edge-detected image. Figure (5) shows the gradient cost.

III. Dynamic Programming Snake

A third technique for making active contours, using dynamic programming, was compared to the other approaches. Unlike the variational calculus method (and similar to the newer greedy method), it operates in the discrete domain: pixels. The consequently simplified numerical methods improve the accuracy; they involve approximation of low-order derivatives, first and second, whereas the calculus method involves up to the fourth.

Dynamic programming differs from the greedy method in that it searches for *globally* optimal solutions as it steps across each snake pixel (henceforth, snaxel). In the greedy

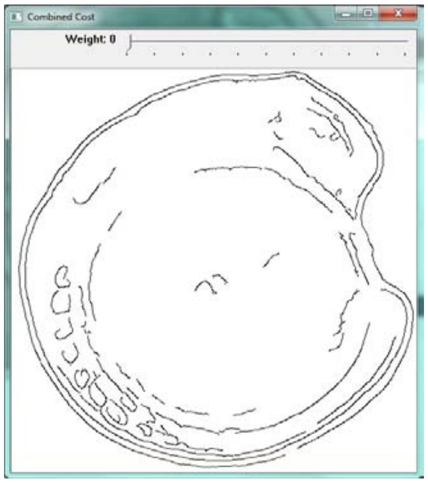

Figure 4. Canny Edge

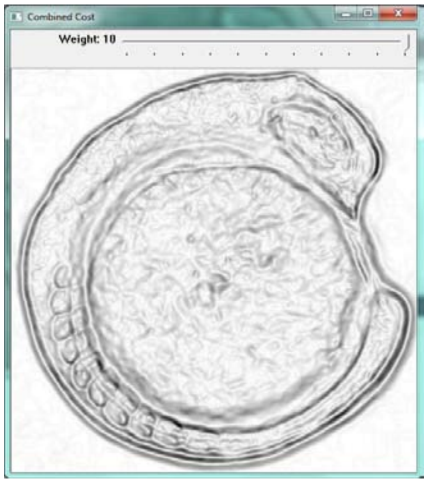

Figure 5. Gradient

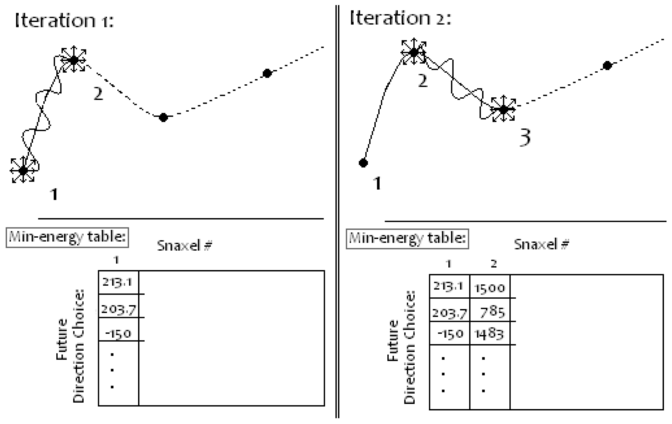

Figure 6. The first couple of iterations. For snaxel 1 (left pane), iterate through all possible neighbors of snaxel 2, and for each of those assumptions calculate the most energy-optimal decision for snaxel 1. That is, stretch the first spring each of the possible ways; we're doing this since we don't know yet what the snaxel 2's decision will be. Store the result of each assumption in the energy table's first column. Now move on to iteration 2 (right pane). Consider as assumptions all possible arrangements of snaxel 3, and calculate the energy of the optimal snaxel 2 decision using each assumption about snaxel 3, this time storing the results in the second column; however, this time, the energy of prior snaxels up to that point (namely snaxel 1) must also be added in. This is something we already stored in the first column; now that we *do* know which decision of snaxel 1 we're considering, we can refer to the row in column one that used the correct assumption, and re-use that calculated value.

method, locally-optimal decisions were simply made for the current snake pixel without ever taking the whole snake into consideration at once.

The implementation in [4] uses textbook dynamic programming. For this that means growing an optimal solution out of smaller optimized subproblems with fewer snaxels. For each new snaxel in consideration it finds the best choice of neighboring [1] locations for it to move into, in order to *globally* minimize the supplied energy equation [2] of the whole snake so far. Once the globally best set of decisions is found for the whole snake, all the snaxels make their move simultaneously, and that constitutes one interation.

More specifically - as we iterate through the snaxel list, assume that the globally optimal energy at the current snaxel will equal the following: First the energy cost of the *locally* best decision for that snaxel, plus the total energy of the entire snake up to that snaxel - but in some snake configuration that is *optimal assuming the local decision that we're considering will definitely happen*. In other words, all the possible next moves must be accounted for at each step.

Rather than repetitively recalculating the energy of all prior snaxels, a table of prior energies is maintained for later consultation; its cells are divided up in a way that covers all the possible decisions that could be taken next.

A. Example

As a first example, we will take the simple case of the membrane term only in the snake equation, without the thin plate term - so only the first derivative approximation is needed. This first-order energy term will involve a finite-difference between the current snaxel and the next one (so, the energy resembles a spring between the two snaxels). To find the amount of energy in this spring, every possible combination of next locations for *both* of these snaxels must be tested since the spring stretches differently if either one moves. So, two iteration loops are needed for checking neighbors: An outer one for perturbing the next snaxel, and an inner one for perturbing the current snaxel.

[1]Neighborhoods are defined arbitrarily in code as an array of relative positions. It is possible to use a neighborhood larger than 3x3, but enlarging the neighborhood comes at an exponential runtime cost increase as will be seen in the final runtime formulas.

[2]Here, the supplied energy equation (aside from the previously explained membrane and thin-plate physics terms) involves the gradient and intensity images. Each snaxel's proximity to certain features in the image helps to guide it, and the features push the snake along until it settles upon the boundary of an object. The gradient image attracts the snake to edges; to help attract it from a distance, this image begins as blurred and, every few iterations, becomes less blurred. This gaussian blur alters the scale space of the features in the image, so that the most severe transitions blur out the farthest and attract the snake at first, and then more local features are allowed to take precedence.

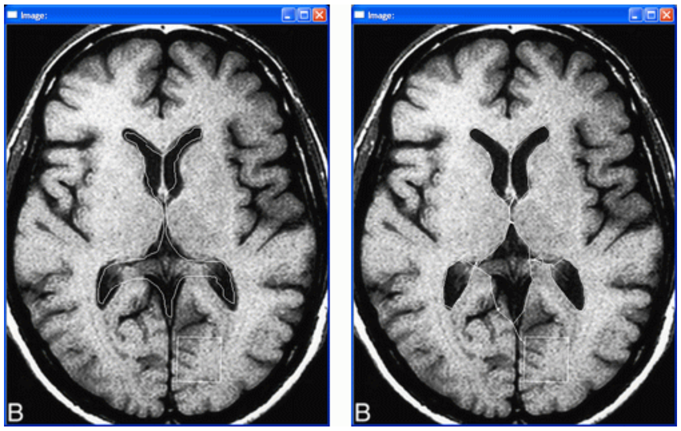

Figure 7. The convergence of a snake to the dark area at the center of a brain image. The initialization is shown on the left.

Associated with each table cell in the energy table is a vector describing exactly *which* of the current snaxel's movements was found to be optimal for that choice; the information would otherwise be lost (since a cell only makes a statement about the next snaxel's decision). To accomplish this we used a "position matrix" that corresponds to our "energy matrix" as suggested in [4]. At the end of iteration, this table of positions can be traced through to infer, in reverse, which decisions led to the best overall energy in the final column (ie. the optimum global state including all snaxels); this decision set describes the simultaneous movement our snaxels should make at the end of this iteration.

B. *Thin Plate Term*

In our implementation the thin-plate snake term was additionally implemented - in fact, all of [4] was implemented, with the sole exception of the image energy term that specifically attracts the snake towards line terminations.

The use of thin-plate term, which influences the snake to try to straighten out as a metal plate or rod would, requires a second order finite-difference approximation of the derivative; that is, a measurement involving the current, previous, *and* next snaxels are needed to estimate curvature. Thus, consideration of three snaxels is needed for each decision.

The example described previously demonstrated the use (as recommended by [4]) of a time-delay in decision-making, wherein a final decision about a snaxel's movement is only made tentatively, until the *next* snaxel's energies are calculated. At any current snaxel we thereby have enough available information to consider the two surrounding snaxels (including the next).

The addition of the thin-plate term complicates the algorithm in that we now need three iteration loops, not two, to fully consider all possible arrangements of the three snaxels centered at the current one; in our implementation the inner-most loop finds the optimal movement of the previous node assuming that the decisions to move the current *and* next nodes go each of their possible ways.

The energy table to store each of these possibilities must now be 3D, to cover all possible assumptions of both the current and the next snaxel's movement when deciding the optimal way to move the previous one. Likewise, the table of positions must become a 3D matrix of vectors. Corner cases (such as the case of there being no previous or next node) must be handled separately. With that, the procedure is ready to emulate a thin-plate. The runtime is O(nm^3) due to the nested iteration loops, and the space requirements are O(nm^2) due to storing the 3D matrix.

One convenience of using the dynamic programming or greedy snake methods is that they accommodate arbitrary hard constraints outside of the energy formula, unlike the original variational calculus method. In our implementation of the dynamic programming snake, hard constraints were added for keeping snaxels a Euclidean distance of two pixels apart, as well as for forcing them to stay within image bounds. This was done simply by rejecting neighbors from consideration for a snaxel to move to if the constraint is violated. The variational calculus approach would require such constraints to be formulated into the energy functional, something that would presumably not be nearly as straight-forward.

IV. Greedy Algorithm Snake

By 1992, three main techniques for implementing the snake model had emerged: the orignal variational method[2], the dynamic programming implementation[4] just described, and the newly introduced and fastest method: the greedy approach[1].

Like in dynamic programming, the greedy snake uses points that move on a discrete grid. Its proposal and subsequent widespread use in an age of slower CPUs than those of today was due to the fact that greedy snakes require little memory and less time to converge, while preserving the advantage of accomodating hard constraints.

A. *Curvature Estimation*

A greedy snake is described with a list of 2D points rather than an implicit curve function. Furthermore, since it uses the greedy algorithm, each point in the list will make a decision after an iteration. For these reasons, effective estimation of curvature becomes an important requirement, and different ways of estimating curvatures should be discussed.

A basic way is to use finite differences, i.e.,

$$\left|\frac{d^2\mathbf{v}_i}{ds^2}\right|^2 \approx \left|\mathbf{v}_{i-1} - 2\mathbf{v}_i + \mathbf{v}_{i+1}\right|^2 .$$

There is a disadvantage for this method: If points are not evenly spaced, this formula will cause the snake to shrink.

A second method is to directly use the mathematical definition of curvature: $d\theta/ds$. Here θ is the angle between the x-axis and the tangent vector to the curve. A discrete approximation for this formula is

$$\Delta\theta = \cos^{-1} \frac{\mathbf{u}_i \cdot \mathbf{u}_{i+1}}{|\mathbf{u}_i|\,|\mathbf{u}_{i+1}|},$$

where $\mathbf{u}_i = \mathbf{v}_i - \mathbf{v}_{i-1}$. This method will necessarily give a satisfying result, but is computationally expensive.

Another method is based on normalizing the two vectors before taking the difference, thereby removing the length differential so that the curvature will only depend on relative direction. This method has some special applications such as detection of corners.

B. Greedy Algorithm

The greedy snake algorithm has a time complexity of O(mn), faster than the O(mn^3) needed by the dynamic programming method, where n is the number of discrete points and m is the size of each point's neighborhood. This algorithm does not guarantee a global minimum energy solution. However, we will show several images for which it acheives satisfactory results.

The idea of this algorithm is quite simple compared to those in previous sections. During each iteration, the neighborhood of each point is examined and the point in the neighborhood giving the smallest value for the energy term is chosen as the new location of the point. Here if we don't consider external forces,

$$E^*_{snake} = \int_0^1 \alpha E_{cont}(\mathbf{v}(s)) + \beta E_{curv}(\mathbf{v}(s)) + \gamma E_{image}(\mathbf{v}(s)) ds.$$

Of course, for each discrete point,

$$E_i = \alpha E_{cont}(\mathbf{v}(s)) + \beta E_{curv}(\mathbf{v}(s)) + \gamma E_{image}(\mathbf{v}(s)).$$

For the continuity term, a finite difference representation causes the curve to shrink. Therefore, $\overline{d} - |\mathbf{v}_i - \mathbf{v}_{i-1}|$ is used instead. Here the average distance between two points is recalculated after each iteration. Since the continuity term has already caused the points to be evenly spaced, one can use the basic finite difference method to estimate the curvature energy. The image strength is again calculated using the image gradient.

After moving the snake during each iteration, an additional step could be processed for corner detection. This is done by setting $\beta = 0$ for a particular point if all of the following conditions are satisfied at the end of each iteration:

1) Curvature is larger than that of both previous and next points.
2) Curvature is above a threshold.
3) Edge strength is above a threshold.

In order to give more user interaction, we have added a feature called "eraser" for the greedy snake. As in Figure (8), if some part of the snake fails to converge to desired edge, the user could just erase them.

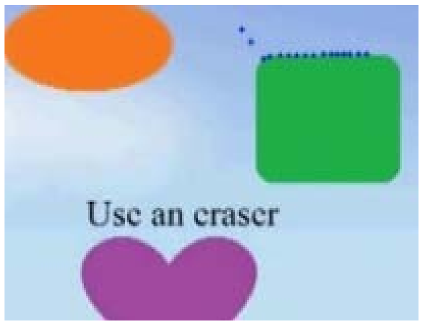

Figure 8. Using an eraser to erase bad points.

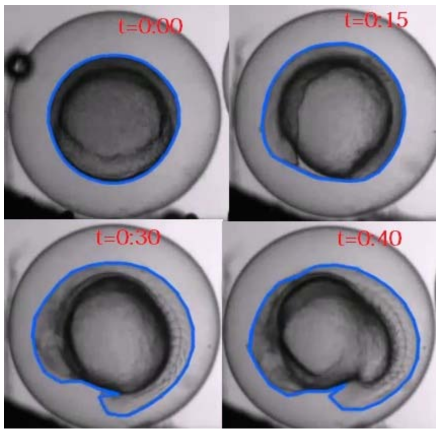

Figure 9. Tracking the embryo fish development with Greedy Snake

V. Video Contour Tracking

The idea of tracking object contours in videos is quite simple. First, the user interactively initilizes a contour on the first frame of video. This could be done with any of the three techniques. After that, our software processes the video frame by frame. During each step, the snake fully converges, and then passes its position on as the initialization for the next frame.

Figure (9) shows the result of tracking a fish embryo's development using a Greedy Snake. The result shows that Greedy Snake performs very well in these sorts of biological applications. In our software interface (Figure 10), the user could make custom configurations for different videos. For example, for faster moving objects, it is naturally neecssary to increase the neighborhood size, so that the snake will not loose its target.

Figure (11) shows the result of tracking a deformable egg with dynamic programming. The original video is a synthetic scene generated by our lab-mate Kresimir. The dynamic programming snake is less sensitive to the initialization than the greedy snake. For that reason, in our software the first-frame initialization for the greedy snake is done by intelligent scissors.

VI. Future Work

One potentially useful application of these types of image segmentation comes from astrophysics. NASA funds

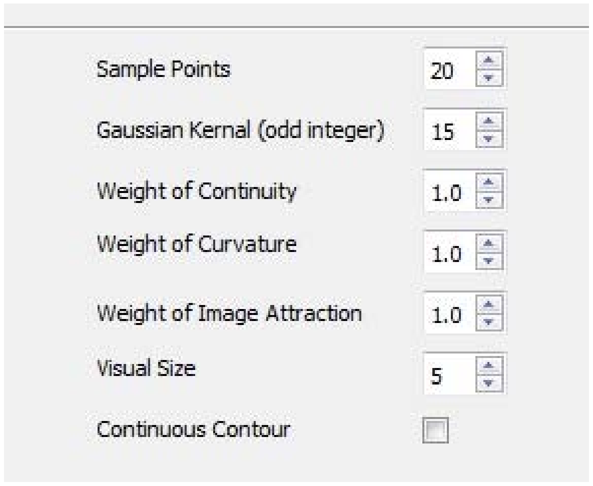
Sample Points 20
Gaussian Kernal (odd integer) 15
Weight of Continuity 1.0
Weight of Curvature 1.0
Weight of Image Attraction 1.0
Visual Size 5
Continuous Contour ☐

Figure 10. Configuration of Greedy Snake

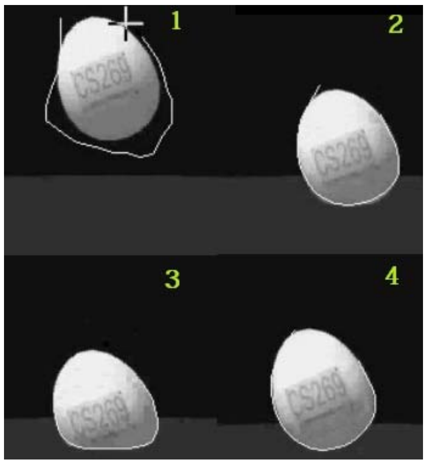

Figure 11. Tracking an deformable egg with Dynamic Programming Snake

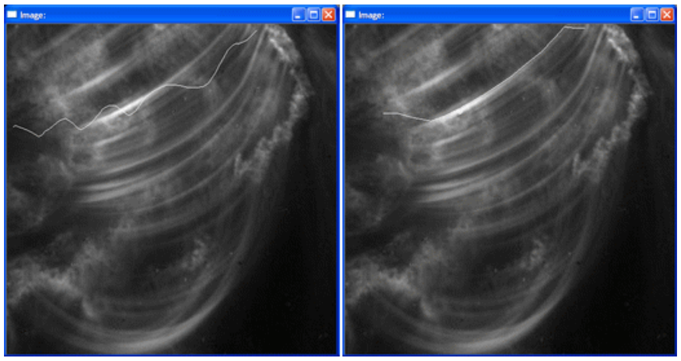

Figure 12. A close-up image of the sun featuring a coronal loop, shown at the beginning and end of snake iteration. Even a poor initialization leads the snake to come to rest along the loop; however, a section of the snake may come to rest on the wrong loops rather than making the jump to the right one. Employing multiple snakes, capable of repelling each other within the image, might solve this problem.

research that studies the mysterious temperature properties of coronal loops emitted from the sun, since they are instrumental in predicting solar storms that can disrupt satellite communication. The work in [8] and [9] shows some attempts to automatically segment and detect solar loops in an image. Figure (12) shows such an image.

Many attempts at segmenting solar images do well at obtaining a subset of the image containing only loop pixels, but these pixels tend to be disjoint, dividing loops into many small segments; cohesion between segments belonging to the same solar loop is often poor. Snakes may provide a good way to isolate an entire loop shape at once.

Loops tend to be shaped like arcs of an ellipse (often overlapping or stacked closely); the thin-plate term of the snake helps to ensure constant curvature as found in an ellipse. The ability to add hard constraints using dynamic programming or the greedy approach might also be helpful, since a penalty imposed when the snake contains an inflection point might help to enforce the elliptical constraint. One remaining challenge, described in [9], is to deal with the overlapping and stacked nature of loops by forcing snakes to lock on to only one solar loop (rather than bridging two nearby ones). Figure (12)'s caption suggests a snake-based solution to this problem as well. Some very recent attempts to use snakes for solar image processing are described in [10] and [11], but the discrete approaches described here may achieve better results in less time than these, especially when combined with the aforementioned scale-space adjustments for attraction to features from a distance.

VII. Conclusion

We have demonstrated software for image segmentation and video contour tracking that uses intelligent scissors and snakes to supplement each other's limitations, and experimented with combining them so that the scissors provide a nice initialization for the snake. We designed a practical GUI so that users could conveniently use the software. It was shown that as long as user finds an appropriate configuration, our software can deliver satisfactory results on biology images and videos, and it shows promise in medical applications. Finally, we have discussed an immediate potential application of this work to the astrophysics problem of automatically tracing the overlapping and closely-stacked coronal loops in images of the sun.

References

[1] D. Williams and M. Shah, A fast algorithm for active contours and curvature estimation, Computer Vision, Graphics, and Image Processing: Image Understanding 55, 1992, 14-26

[2] Kass, M., Witkin, A., and Terzopoulos, D., "Snakes: Active Contour Models", International Journal of Computer Vision, 1(4):321–331, 1987

[3] Menet, Saint-Marc and Medioni, B-Snakes: Implementation and Application to Stereo

[4] A. Amini, S.Tehrani, and T.Weymouth, "Using dynamic programming for minimizing the energy of active contours in the presence of hard constraints," in Proc. Second Int. Conf. Computer Vision, Tarpon Springs, FL, Dec. 1988

[5] E. N. Mortensen and W. A. Barrett, "Intelligent Scissors for Image Composition," in Computer Graphics (SIGGRAPH ‘95), pp. 191-198, Los Angeles, CA, Aug. 1995

[6] http://www.cs.ucla.edu/~petrinec/

[7] E.W.Dijkstra, A note on two problems in connexion with graphs. Numerische Mathematick 1, 269-271, 1959

[8] Nurcan Durak, Olfa Nasraoui, Joan Schmelz, Coronal loop detection from solar images, Pattern Recognition, v.42 n.11, p.2481-2491, November, 2009

[9] J. Lee, T. S. Newman, and G. A. Gary, "Oriented Connectivity-based Method for Segmenting Solar Loops," Pattern Recognition, Vol. 39, 2006, pp. 246-259

[10] Gill, C. D.; Fletcher, L.; Marshall, S; “Using Active Contours for Semi-Automated Tracking of UV and EUV Solar Flare Ribbons “, Solar Physics (2010) 262: 355-371, March 22, 2010

[11] Tang, Woon Khang, “Automated Coronal Loop Segmentation Using Snake-Based Algorithm”, http://rave.ohiolink.edu/etdc/view?acc_num=bgsu1277139373