

CS 312 Project 1 - Intelligent Scissors due January 30 2006

1/26/2006 10:32 PM

Background

“Intelligent Scissors” [1] is an interactive image segmentation technique that uses Dijkstra’s algorithm to find minimal cost boundary paths in an image. The intelligent scissors algorithm works as follows:

- The user clicks on a “seed point” somewhere along the desired segmentation boundary.
- Dijkstra’s algorithm is used to find a minimal cost path from the seed point to every other pixel in the image.
- The user moves the cursor to a location further along the segmentation boundary. As the cursor moves, the minimum cost path back to the seed point is displayed.
- The user clicks on the image again, and the minimum cost path back to the seed point is added to the segmentation boundary.
- Steps 2 to 4 are repeated using the new point as the seed until the original seed point is chosen again, making a closed boundary.

Intelligent Scissors works well in practice and is the basis of a number of segmentation algorithms in commercial image editing software. For example, the right image below was segmented with just 3 seed locations. Contrast this with a simple greedy algorithm that just follows the least cost edge at each node. The simple algorithm adheres to the edge in this case, but passes right by the point it is supposed to meet up with (near the eye socket of the skeleton) and then gets stuck on the cheek bone of the skeleton. For this assignment, you will be programming both a “Simple Greedy” scissors and “Intelligent Scissors” using Dijkstra’s algorithm.



Simple Greedy



Dijkstra’s Algorithm

Images as Graphs

An image can be treated as a graph by considering each pixel in the image to be a node in the graph. Edges in the graph are connections between pixels. For this assignment, we will assume that pixels are only connected to their vertical and horizontal neighbors. The weight of an edge going from one pixel to another will be defined as follows:

$$\text{Weight} = 1 + MG - G$$

G = the gradient magnitude for the destination pixel

MG = the maximum gradient magnitude for any pixel in the image

To compute the gradient magnitude of pixels in the image, we will use an operator called the “Sobel Filter” that calculates a discrete approximation of the x and y partial derivatives of the image. Using the Sobel filter, the gradient magnitude (G_{xy}) for the pixel P_{xy} , is defined as follows:

$$GX_{xy} = ((P_{x+1,y+1} - P_{x-1,y+1}) + 2*(P_{x+1,y} - P_{x-1,y}) + (P_{x+1,y-1} - P_{x-1,y-1})) / 4$$

$$GY_{xy} = ((P_{x+1,y+1} - P_{x+1,y-1}) + 2*(P_{x,y+1} - P_{x,y-1}) + (P_{x-1,y+1} - P_{x-1,y-1})) / 4$$

$$G_{xy} = \sqrt{GX_{xy}^2 + GY_{xy}^2}$$

Working with images

We have included a VisualStudio 2005 project that contains a class for manipulating ASCII pgm images (we have also some pgm images).

The project now (1/26) includes pretty much everything except implementations of the Dijkstra-based and simple greedy algorithms. Here’s a few pointers:

- `Image.Bitmap.SetPixel` is the method you’ll want to use to set the color of a pixel.
- After you set a pixel, you need to tell the image to update. You can use `Program.MainForm.Refresh();` to do that. And, you can update as many pixels as you want between updates and they’ll all get updated.
- `DijkstraScissors.cs` and `SimpleScissors.cs` are the files that contain the classes you want to implement. The method `Trace` in each file is the entry point for the class when the appropriate button is clicked. The parameters are explained in the file.
- Feel free to use any built in datastructure you like and feel free to download a priority queue class if you don’t want to write your own.

In case you are using the original files, the project originally included

- A form (`Form1.cs` and `Form1.cs [Design]`) for opening and viewing pgm images. The form also includes a button for calculating the gradient over the entire image.
- A class called “`grayImage`” which includes the following methods:

- `load (s:string)` load an image out of a file with name `s`. Right now, it is hardcoded to load ASCII images only.
- `loadASCII (s:string)` loads an ASCII pgm image and returns a bitmap. As a side effect, the gray values for every pixel in the image are stored in the “bitmap” property of the `grayImage` class. WARNING: slow but could be faster.
- `gradient (x:int, y:int)` calculates and returns the gradient for pixel at location `x,y`.
- `calculateGradientImage` which calculates the gradient for the entire image. WARNING: slow.
- `setPixel (x: int, y:int, gray:int)` sets pixel `x,y` to gray value `gray`. In the distributed code, this is a blatant violation of model-view-controller that will actually cause your algorithm to perform incorrectly. So change it (but you don’t have to go all the way to pure model-view-controller if you don’t have the time or inclination.)

What to do

Your assignment is to code up an image segmentation algorithm based on finding the least-cost path between selection points in an image. The definition of the cost of moving between adjacent pixels is given in the “Images as Graphs” section above.

You will be given the image in an ASCII pgm file. PGM files allow comments. You will be given an image that includes the selection points in a comment in the second line of the image file. The segmentation points will be given in the order in which they were selected by the user. The segmentation points will be given in a semi-colon separated list of tuples and the tuples will consist of parentheses around a pair of integers separated by commas. For example, your image file might look something like this:

```
P2
# (30,40); (50,60); (10,20); (50,80)
# Created by Ifranview
384 256
255
etc.
```

After reading the image and segmentation points, your algorithm should

1. Use either the Simple Greedy scissors or Dijkstra’s algorithm to compute the shortest path between adjacent pairs of points. This is called the “segmentation path”. You’ll need to do both implementations. The project distribution includes a drop-down menu in which one can select the segmentation algorithm. You can implement the methods for this interface or design your own. See Part 1 and Part 2 below for a description of each algorithm. Remember that the segmentation path is a cycle that ends at the starting segmentation point.

2. Set the color of each segmentation point to white or red or some other color.
3. Set the color of each pixel on the segmentation path to white or red or some color other than a dark shade of gray or black.
4. Output how long it took to do steps 1-3 into the textbox on Form1 or some other place of your choosing (as long as it can be seen on the screen).
5. For the writeup, you will need to put a copy of the resulting image into your writeup and report the running time of the selected algorithm. You can write the image to a file (note that PGMs allow only grayscale) or try to capture the screen. Up to you. The point of this is that looking at the resulting segmentation gives one a good idea of how good the segmentation is. So, we want to be able to see what you end up with and compare the results.

Part 1: “Simple Greedy” Scissors

For the first part of the assignment, you will be programming a simplified Scissors method that does not use Dijkstra’s algorithm. Instead, at each node in the graph, simply choose the edge that

1. has the smallest weight and
2. does not form a cycle in your path

until the goal is reached, or the path cannot continue. See the section “Images as Graphs” for a description of how to treat images as graphs.

Part 2: “Intelligent” Scissors Using Dijkstra’s Algorithm

For the final part of the assignment, you need to implement a basic version of Intelligent Scissors that uses Dijkstra’s algorithm to find the optimal paths between pixels. Pseudocode for Dijkstra’s algorithm can be found on page 199 of the textbook, and further explanations can be found in the lecture slides.

.

Part 3: Comparison and Performance

Compare and contrast the simple and intelligent scissors by attempting to solve three different segmentation problems. At least two must come from the “with-segmentation-points” subdirectory of the “sample images” directory in the project file distribution. The other one can come from anywhere you like, including the “with-segmentation-points” subdirectory.

Turn in a hardcopy of the following report to the TAs:

1. Cut and paste the resulting image files onto a page to highlight the difference in performance. Include the running time for each algorithm and each image. Turn in a hardcopy of this page to your TA when you pass off your code.
2. Write a few sentences about the trade offs between running time and precision that you observe in your experiments.
3. Write a few sentences to answer the following question: This algorithm could be used in tools like Adobe Photoshop or the GIMP. How would you modify your algorithm to make it fast enough to run interactively?
4. Say whether or not you met the performance requirement. The performance requirement is that one of your algorithms should solve segment each of the images you chose in 5 seconds or less
5. Include a copy of your method or class that does Dijkstra's algorithm. This is so that we can verify that you actually used Dijkstra's algorithm. If you are keen to create your own specialized shortest path algorithm, then do that for the extension.

Improvement (due February 1 2006)

Think of a way to improve your algorithm and implement it. Determine if your implementation had the desired effect. Write a page or two (1 page is really ok, so don't feel obligated to go long if you finish saying what you want to say in a page) that describes your implementation and what it is supposed to do to improve the algorithm. Then, give some empirical data to support a conclusion that your improvement either did or did not do what you thought it would do.

Here are a few suggestions for improvements:

- Make the intelligent scissors interactive. That is, allow the user to click on points and then find the segmentation path between those points.
- Support color images. You'll need to pick a new image format (and probably want to use Microsoft's built-in tools for loading images) and need to come up with a gradient equation that accounts for color.
- Improve the running time of one of your segmentation algorithms by improving the implementation. This might include using an algorithm other than Dijkstra's algorithm.
- The "compute gradient over the entire image" method is woefully slow. Make it faster.
- Add a view of your segmentation results that superimposes the segmentation paths on top of the gradient view of the image. Add a button to zoom in on the image so you can get a better idea of whether or not your algorithm is following the gradient like you think it does.
- Render the segmented view of the image as a height-field so that it looks like a 3-dimensional terrain. Then, draw your segmented path on the terrain and see if the algorithm is following the gradient like you think it does. DirectX9 would probably be useful for this.
- Reduce the amount of memory that your algorithm consumes.

- Make up your own improvement based on something that interests you and is related to the project. Talk to your professor if you aren't sure if your idea qualifies.

Reference

[1] 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.