Problem Set 3
CMSC 426
Assigned: Feb. 23, 2010; Due: Mar. 9, 2010

Synopsis

In this project, you will create a tool that allows a user to cut an object out of one image and paste it into another. The tool helps the user trace the object by automatically following the boundary of the object of interest. You will then use your tool to create a composite image.

Forrest Gump shaking hands with J.F.K.

You will be given some Java functions that provide the user interface elements and data structures that you'll need for this program. These are described below

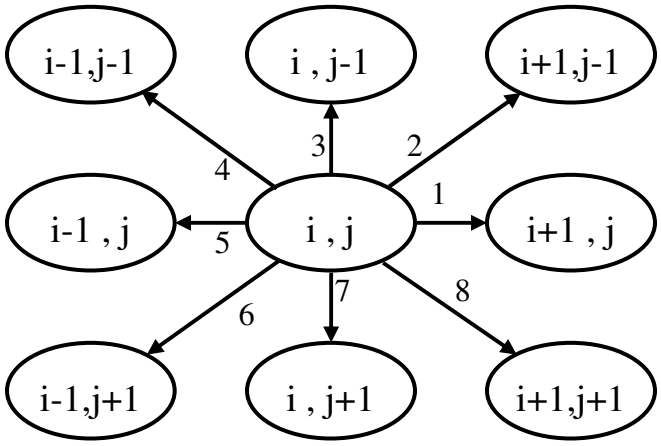
Description

This program is based on the paper Intelligent Scissors for Image Composition, by Eric Mortensen and William Barrett, published in the proceedings of SIGGRAPH 1995. You will implement a simplified version of the program described in that paper. The way it will work is that the user first clicks on a "seed point" which can be any pixel in the image. The user then clicks on a second point in the image. The program then computes a path from the seed point to the second point that hugs the contours of the image as closely as possible. This path, called the "live wire", is computed by converting the image into a graph where the pixels correspond to nodes. Each node is connected by links to its 8 immediate neighbors. Note that we use the term "link" instead of "edge" of a graph to avoid confusion with edges in the image. Each link has a cost relating to the derivative of the image across that link. The path is computed by finding the minimum cost path in the graph, from the seed point to the mouse position. The path will tend to follow edges in the image instead of crossing them, since the latter is more expensive. The path is represented as a sequence of links in the graph. The user continues, clicking point after point to map out the outline of an object. In the end, the user closes the path by clicking on a point near the first, seed point, and a connection is found between the

final clicked point and the seed point. The object encircled by this path is then extracted from the image.

Next, we describe the details of the cost function and the algorithm for computing the minimum cost path. The cost function we'll use is a bit different than what's described in the paper, but closely matches what is discussed in lecture.

As described in the lecture notes, the image is represented as a graph. Each pixel (i,j) is represented as a node in the graph, and is connected to its 8 neighbors in the image by graph links (labeled from 1 to 8), as shown in the following figure.



Cost Function

To simplify the explanation, let's first assume that the image is grayscale instead of color (each pixel has only a scalar intensity, instead of a RGB triple) as a start. The same approach is easily generalized to color images.

- Computing cost for grayscale images

  Among the 8 links, two are horizontal (links 1 and 5), two are vertical (links 3 and 7), and the rest are diagonal. The magnitude of the intensity derivative across the diagonal links, e.g. link2, is approximated as:

  ```
  D(link 2)=|img(i+1,j)-img(i,j-1)|/sqrt(2)
  ```

  The magnitude of the intensity derivative across the horizontal links, e.g. link 1, is approximated as:

```
D(link 1)=|(img(i,j-1) + img(i+1,j-1))/2 - (img(i,j+1)
                + img(i+1,j+1))/2|/2
```

Similarly, the magnitude of the intensity derivative across the vertical links, e.g. link 3, is approximated as:

```
D(link 3)=|(img(i-1,j)+img(i-1,j-1))/2-
    (img(i+1,j)+img(i+1,j-1))/2|/2.
```

We compute the cost for each link, cost(link), by the following equation:

```
cost(link)=(maxD-D(link))*length(link)
```

where maxD is the maximum magnitude of derivatives across links over in the image, e.g., maxD = max{D(link) | forall link in the image}, length(link) is the length of the link. For example, length(link 1) = 1, length(link 2) = sqrt(2) and length(link 3) = 1.
If a link lies along an edge in an image, we expect that the intensity derivative across that link is large and accordingly, the cost of link is small.

- Cost for an RGB image

  An RGB image is stored in Java as an nxmx3 array. The third dimension contains the red, blue and green components of the image, respectively. For example, if I is an RGB image, then I(:,:,1) contains the red in the image. As in the grayscale case, each pixel has eight links. We first compute the magnitude of the intensity derivative across a link, in each color channel independently, denoted as

  ```
  ( DR(link),DG(link),DB(link) ).
  ```

  Then the magnitude of the color derivative across link is defined as

  ```
  D(link) = sqrt(
  (DR(link)*DR(link)+DG(link)*DG(link)+DB(link)*DB(link)
  )/3 ).
  ```

  Then we compute the cost for link link in the same way as we do for a gray scale image:

  ```
  cost(link)=(maxD-D(link))*length(link).
  ```

Notice that cost(link 1) for pixel (i,j) is the same as cost(link 5) for pixel (i+1,j). Similar symmetry property also applies to vertical and diagonal links.

- Using cross correlation to compute link intensity derivatives

You should compute the D(link) formulas above using 3x3 correlation or convolution. Each of the eight link directions will require using a different kernel. You will need to figure out for yourself what the proper entries in each of the eight kernels will be. You can do this using Java's ConvolveOp. There is an example in the code we are giving you showing how to use this to compute a derivative.

Computing the Minimum Cost Path

The pseudo code for the shortest path algorithm in the paper is a variant of Dijkstra's shortest path algorithm, which is described in any algorithm text book (e.g., Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein, published by MIT Press). Here is some pseudo code which is equivalent to the algorithm in the SIGGRAPH paper, but we feel is easier to understand.

function find_shortest_path

input: array[][][],*seed*, *goal, image_height,image_width*
array[][][] stores the eight cost of links for all of the pixels.

Output: A two dimensional array with the first point being goal and the last point being immediately next to the seed. Keep the array size constant with Iscissor.SIZE_MAX, put the points on the path at the beginning of the array, and fill in the rest of the array with -1.

comment: each node will experience three states: INITIAL, ACTIVE, EXPANDED sequentially. The algorithm terminates when the goal-point has been extracted from the queue, so that the shortest path to it is known. All nodes in the graph are initialized as INITIAL. When the algorithm runs, all ACTIVE nodes are kept in a priority queue, *pq*, ordered by the current total cost from the node to the seed.

Begin:

initialize the priority queue *pq* to be empty;

initialize each node to the INITIAL state;

set the total cost of *seed* to be zero;

insert *seed* into *pq*;

extract the node *q* with the minimum total cost in *pq*;

% Now find the shortest path.

while *q* is **not** (0,0) & q is not goal

% If *q* is (0,0), it means the queue is empty.

mark *q* as EXPANDED;

for each neighbor node *r* of *q*

if *r* has **not** been EXPANDED

mark *r* as ACTIVE;

insert *r* in *pq* with the sum of the total cost of q and link cost from *q* to *r* as its total cost;

if inserting r changed it

make an entry for *r* in the Pointers array indicating that currently the best way to reach *r* is from *q*.

extract the node *q* with the minimum total cost in *pq*;

End

% Trace back your solution.

Initialize *wire* array to be zeros.

Set current pixel to the goal pixel.

Set current pixel in *wire* to be 1

While current pixel ~= seed pixel

Set current pixel to be predecessor to current pixel retrieved from Pointers

set current pixel in *wire* to be 1

End

Return wire

We provide the priority queue functions that you will need in the skeleton code (implemented as an unsorted array). These are: **Create**, **extractmin** and **insert**.

Skeleton Code

You can download the supporting files from the class web page. Here is a description of what's in there:

- Iscissor.Java : This is the main class and contains the main() method in it. It contains a number of methods but from them we list a few below.

getDerivativAarray(BufferedImage img,int hgt,int wdth)

This method takes input an image, its height and width and returns a 3-dimensional double array of size height*width*8. For each of the points in the image array, we have to store the cost of the eight links connected to it.

livewire(BufferedImage img,int img_height,int img_width)

This method takes the image, height, width as input and returns a 2-dimensional array. It calls the Shortest_Path method in it. It returns the boundary points of the selected area.

copypaste(BufferedImage img,BufferedImage imgc,int img_height,int img_width, int live_wire[][])

This method takes the image, the image where it would copy,height and width of original image and the 2-dimensional array with all the boundary points of the selectede area. It basically does the pasting part and this method does not return anything.

- Short_Path_modified.Java : In this class the shortest path algorithm will be implemented. Start[] and end[] are one dimensional arrays of size two. The first field holds the X coordinate and the 2nd field holds the Y coordinate. Arr[][][] is a 3-d array of size height*width*8 and holds the link cost of eight links around the pixels in the image.

- Linear_path.java : This class has been implemented and it finds a straight linear path between two points.

- MyImagePanel.java : This class helps in display and resizing of the image and has been implemented.

- Point.java : This class implements a point object (with cost comparator) and can be used to implement a priority queue.

- Test_q.java : This class has CREATE(), INSERT() and EXTRACTMIN() methods, which are to be used to implement the PriorityQueue.

Steps to use this code:

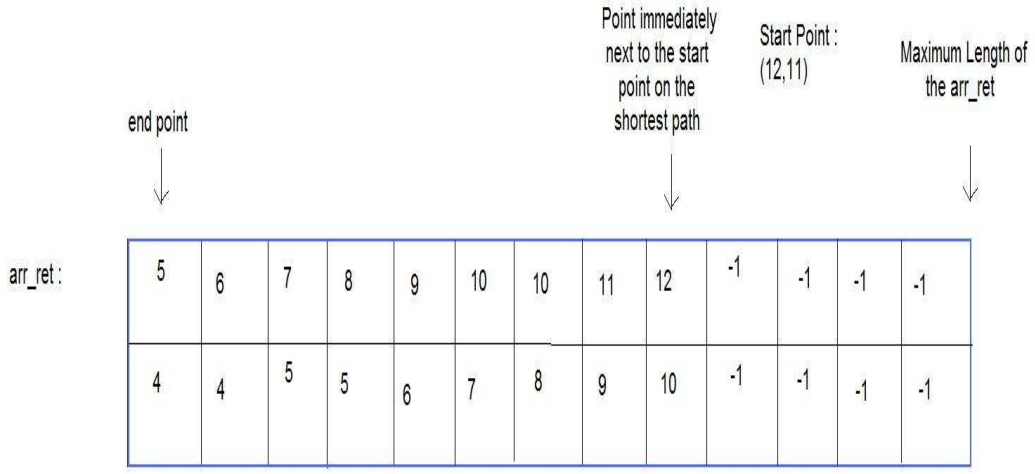
When you will implement the required methods properly, the code should work in the following way. Please change the name of the directory of testing images based on your machine in the readimage() method.

1. Run Iscissor.java.
2. Enter the file name from which you want to crop.
3. Enter the file name in which you want to paste.
4. The first file will open and you would select a point (first point) by left clicking on the window. And you will left click again (second point) and you see the boundary formed using the shortest path algorithm. If you do not like that boundary right click on the image and again left click where you want to place the second point.
5. You would continue doing this and keep the last point of your selection very close to the beginning point. Then left click anywhere in the image and the image where you want to paste will open. You can select maximum 20 points to determine a boundary.
6. Left click anywhere to place that (Be careful such that the cropped portion fits in there). If you want change the position of pasting do that by left clicking on different parts of the new image window. Finally right click anywhere to complete your job.
7. Save this new image and also the old image with the selected part.

Some guidelines on how to use this Code:

- When you place the cropped portion in the new image be careful so that the part fits in within that image. The cropped image would be placed in the new image around the point where you left click.
- Don't crop a very large image. Keep it small. There should not be any problem as long as you crop images of size around 400*400.
- Try to avoid copying rectangular/square structures. Basically if there are a very large number of horizontal points on the cropping boundary for a fixed vertical point, the code may not work well. We recommend using any irregular shapes for cropping.
- Do not take images very large. Please keep them less than 1000*1000 pixels.
- Please do not lose track of the first point. Keeping the last point very close to the first point is the only way to complete selecting an area.
- Finally, the portions you copy might be a little bit crazy in the new image if the cropped portion is not a convex polygon. Don't worry about that.

What the Shortest path algorithm is supposed to return:



Run-time notes: Our program does not show the path as you move the mouse, but requires you to click on the next point, and then computes the appropriate path. In good circumstances, it will take a few seconds to compute the next stage in your path. However, run-time can vary. Two things can make the program very slow. First, if the image contains a lot of texture and strong gradients, many paths will have a low cost. Exploring all these possible paths will be much slower than when a single strong gradient produces one low-cost path. Second, if you click on two points that are very far apart, finding the best path between them may be quite slow.

To Do

Implement find_shortest_path method

Implement getDerivativAarray method

Use these to create a new, composite image.

The Artifact

For this assignment, you will turn in a final image (the *artifact*) which is a composite created using your program. Your composite can be derived from as many different images as you'd like. Make it interesting in some way--be it humorous, thought provoking, or artistic! You should use your own scissoring tool to cut the objects out.

Hand in:

You should hand in all code that you have written or modified. Also hand in your composite image and the raw images that you used to generate it, along with a brief description of how you did it. Check out some of the projects done by Steve Seitz's class (there is a link to it on the class web page) for inspiration and an idea of useful descriptions of what was done. Email the TA a single tar'ed file with all this information too, including the images you created and used to create them.

Challenge Problems

Here is a list of suggestions for extending the program for extra credit. If you choose to implement one of these, turn in the code you wrote along with a description of what you did. You are encouraged to come up with your own extensions. We're always interested in seeing new, unanticipated ways to use this program!

Modify the interface and program to allow blurring the image by different amounts before computing link costs. Describe your observations on how this changes the results.

Try different costs functions, for example the method described in Intelligent Scissors for Image Composition, and modify the user interface to allow the user to select different functions. Describe your observations on how this changes the results.

Implement code to allow each point clicked on to snap to the nearby point with the highest magnitude of gradient, as described in Intelligent Scissors for Image Composition.

Feel free to come up with your own improvements to the program. Describe anything you do in detail.

Credits

This basic problem set was created by Steve Seitz for his computer vision class at the University of Washington (based of course on the Intelligent Scissors paper). All concepts, and the write-up of this problem set, are heavily adapted from his problem set. Konstantinos Bitsakos ported this problem set to Matlab. Arijit Biswas has ported it to Java.