**Intro to Computer Vision (CS4670/5670), Spring 2015**

**Project 1:  Image Scissors**

|  |
| --- |
| **Assigned**:  Monday, February 2nd **Due:**  Thursday, February 12th (by 09:00 am), via [CMS](http://cms.csuglab.cornell.edu/) **Demos:**  Thursday, February 12th, signup on [CMS](http://cms.csuglab.cornell.edu/) **Artifact Due:** Thursday, February 12th (by 11:59 pm), via [CMS](http://cms.csuglab.cornell.edu/)  **This project should be done in groups of 2 students.** |

[Synopsis](http://www.cs.cornell.edu/courses/cs4670/2015sp/projects/pa1/#synopsis)  
[Description](http://www.cs.cornell.edu/courses/cs4670/2015sp/projects/pa1/#description)  
[Downloads](http://www.cs.cornell.edu/courses/cs4670/2015sp/projects/pa1/#downloads)  
[Skeleton Code](http://www.cs.cornell.edu/courses/cs4670/2015sp/projects/pa1/#skeleton_code)  
[To Do](http://www.cs.cornell.edu/courses/cs4670/2015sp/projects/pa1/#to_do)  
[Handing In](http://www.cs.cornell.edu/courses/cs4670/2015sp/projects/pa1/#handingin)  
[Bells and Whistles](http://www.cs.cornell.edu/courses/cs4670/2015sp/projects/pa1/#bells_and_whistles)

**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 providing a "live wire" that automatically snaps to and wraps around the object of interest.  You will then use your tool to create a composite image and the class will vote on the best composites.

  
**Forrest Gump shaking hands with J.F.K.**

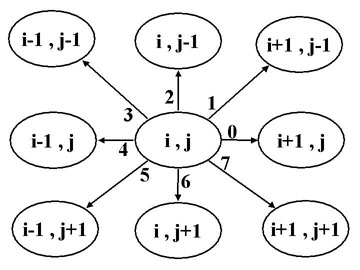
In the [Downloads](http://www.cs.cornell.edu/courses/cs4670/2015sp/projects/pa1/#downloads) section, we provide a virtual machine that has all necessary dependencies, and a working skeleton program which provides the user interface elements and data structures that you'll need for this program.  This skeleton is described below.  We have [provided a sample solution executable and test images](http://www.cs.cornell.edu/courses/cs4670/2015sp/projects/pa1/bin/scissors-solution).  Try this out to see how your program should run. If you have any problem running the solution executable, please use Piazza (recommended) or email a TA.

**Description**

This program is based on the paper [*Intelligent Scissors for Image Composition*](http://www.cs.cornell.edu/courses/cs4670/2012fa/readings/mort-sigg95.pdf), by Eric Mortensen and William Barrett, published in the proceedings of SIGGRAPH 1995.  The way it works is that the user first clicks on a "seed point" which can be any pixel in the image.  The program then computes a path from the seed point to the mouse cursor 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.

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 was 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 0 to 7), 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). The same approach is easily generalized to color images.

* Computing cost for grayscale images

Among the 8 links, two are horizontal (links 0 and 4), two are vertical (links 2 and 6), and the rest are diagonal. The magnitude of the intensity derivative across the diagonal links, e.g. link1, is approximated as:

D(link1)=|img(i+1,j)-img(i,j-1)|/sqrt(2)

The magnitude of the intensity derivative across the horizontal links, e.g. link 0, is approximated as:

D(link 0)=|(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 2, is approximated as:

D(link2)=|(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 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 0) = 1, length(link 1) = sqrt(2) and length(link 2) = 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

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 in the same way as we do for a gray scale image:

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

Notice that cost(link 0) for pixel (i,j) is the same as cost(link 4) for pixel (i+1,j). A similar symmetry property also applies to vertical and diagonal links.

For debugging purposes, you may want to scale down each link cost by a factor of 1.5 or 2 so that they can be converted to byte format without clamping to[0,255]

* Using cross correlation to compute link intensity derivatives

You are **required** to implement the D(link) formulas above using 3x3 cross correlation. Each of the eight link directions will require using a different cross correlation kernel. You will need to figure out for yourself what the proper entries in each of the eight kernels will be. However, this does not mean that you have to entirely compute D(link) using only cross correlation since this is not possible. But you should be able to compute the intensity derivative (not the magnitude) for each of the three channels using only cross correlation.

**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 of the classic algorithm books (including text books used in data structures courses such as 3110 or the [2112 notes](http://www.cs.cornell.edu/courses/CS2112/2014fa/lectures/lecture.html?id=ssp).).  You could also refer to any of the classic algorithm text books (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.  
  
**procedure LiveWireDP**

**input**: *seed*, *graph*  
  
**output**: a minimum path tree in the input *graph* with each node pointing to its predecessor along the minimum cost path to that node from the seed.  Each node will also be assigned a total cost, corresponding to the cost of the the minimum cost path from that node to the seed.  
  
**comment**: each node will experience three states: INITIAL, ACTIVE, EXPANDED sequentially. the algorithm terminates when all nodes are EXPANDED. All nodes in 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*;  
  
while *pq* is **not** empty

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

mark *q* as EXPANDED;  
  
for each neighbor node *r* of *q*

if  *r* has **not** been EXPANDED

if  *r* is still INITIAL

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

mark *r* as ACTIVE;

else if  *r* is ACTIVE, e.g., in already in the *pq*

if the sum of the total cost of q and link cost between *q* and *r* is less than the total cost of r

update the total cost of *r* in *pq*;

**End**

A priority queue is a data structure for maintaining a set of elements, each with an associated key. [Here is a short introduction to priority queues](http://en.wikipedia.org/wiki/Priority_queue).

[We provide the following priority queue functions that you will need in the skeleton code (implemented as a binary heap) (see PriorityQueue.h).](http://www.cs.cornell.edu/courses/cs4670/2015sp/projects/pa1/project1_files/priorityqueue-h.html)

**Downloads**

 **Virtual machine**: We provide you with a VirtualBox [virtual machine](https://cornell.box.com/s/rhth59tk4uxjvqnsxno15w9wwhd9zumh) running Ubuntu with all the necessary dependencies installed (password is 'student'). See this [video](http://www.cs.cornell.edu/courses/cs4670/2015sp/projects/pa1/www.youtube.com/watch?v=WpbUr_fWa7Q) for a tutorial on setting up the VM and the features of the PA1 program.

 **Skeleton code**: Available through [git](http://git-scm.com/) (this should help make distributing any updates easier). With git installed, you can download the code by typing (using the command-line interface to git)

>> git clone http://www.cs.cornell.edu/courses/cs4670/2015sp/projects/pa1/skeleton.git

This will create the directory skeleton. To get updates to the code you can then simply run

>> git pull

This will fetch any updates and merge them into your local copy. If we modify a file you have already modified git will not overwrite your changes. Instead, it will mark the file as having a conflict and then ask you to resolve how to integrate the changes from these two sources. [Here's](http://www.kernel.org/pub/software/scm/git/docs/v1.7.3/user-manual.html#resolving-a-merge) a quick guide on how to resolve these conflicts.

For those that are already using git to work in groups, you can still share code with your partner by having multiple masters to your local repository (one being this original repository and the other some remote service like github where you host the code you are working on); [here's](http://stackoverflow.com/questions/849308/pull-push-from-multiple-remote-locations) a reference with more information. Note that publicly visible git repositories with solution code constitute a violation of academic integrity. So if you're collaborating using Github or similar, you must make your repository private.

 **Solution executable:** [Download](http://www.cs.cornell.edu/courses/cs4670/2015sp/projects/pa1/bin/scissors-solution)

 **Boilerplate .zip archives:** For the [webpage](http://www.cs.cornell.edu/courses/cs4670/2015sp/projects/pa1/downloads/webpage.zip).

**Skeleton Code**

This project uses [FLTK](http://www.fltk.org/) (the Fast Light Toolkit) as a UI framework; it is installed on the VM already. All the source files (.h/.cpp) are in the subdirectory src.  There are 28 files total, but many are just user interface files that are automatically generated by [fluid](http://fltk.org/doc-1.1/fluid.html#10_1), the FLTK user interface building tool.  Here is a description of what's in there:

* ImageLib/\*.cpp/h are files taking care of image file I/O. The only image format we support in this tool is targa (.tga) format, which is also readable in Photoshop and most other image viewing and editing tools.  One reason we like tga is that it's possible to store a transparency (alpha) channel with the RGB image data, in 32 bit mode.  You shouldn't have to change these files.
* IScissorPanelUI.h/cpp, BrushConfigUI.h/cpp, FltDesignUI.h/cpp, ImgFilterUI.h/cpp, and HelpPageUI.h/cpp define four types of windows used in the tool. They are generated by [fluid](http://fltk.org/doc-1.1/fluid.html#10_1) through BrushConfigUI.fl, FltDesignUI.fl, ImgFilterUI.fl, and HelpPageUI.fl. You don't have to worry too much about them if you don't want to change the structures of the windows.  If you do decide to change the UI, you may prefer using fluid to do so.
* imgflt.h is a header file used in most of the files.
* FltAux.h defines a few auxiliary structures/routines for the application.
* ImgFltMain.cpp is the main file where you want to start reading.
* correlation.h/cpp are the files where your image filtering code will go.
* ImgView.h/cpp are files where most of the UI requests are implemented. In ImgView.cpp, you want to finish handle() so that it applies the brush filter when the left mouse button is pressed and moved (as Ctrl is held).
* PriorityQueue.h, which defines several template classes for dynamic array, binary heap, and doubly linked list. They are useful for representing contours and searching the minimum path tree.

ImgView contains most of the data structures and handles interface messages. You will work with iScissor.cpp most often.

**Data structures**

The main data structure that you will use in Project 1 is the Pixel Node.

**Pixel Node**

Use the following Node structure when computing the minimum path tree.

struct Node{  
    double linkCost[8];  
    int state;  
    double totalCost;  
    Node \*prevNode;  
    int column, row;   
    //other unrelated fields;}

* **linkCost** contains the costs of each link, as described above.
* **state** is used to tag the node as being INITIAL, ACTIVE, or EXPANDED during the min-cost tree computation.
* **totalCost** is the minimum total cost from this node to the seed node.
* **prevNode**

points to its predecessor along the minimum cost path from the seed to that node.

* **column** and **row** are used to store the position of the node in original image so that its neighboring nodes can be located.

For visualization purposes, we provide code to convert a pixel node array into an image that displays the computed cost values in the user interface.  This image buffer, called Cost Graph, has the structure shown below.  Cost Graph has 3W columns and 3H rows and is obtained by expanding the original image by a factor of 3 in both the horizontal and vertical directions. For each 3 by 3 unit, the **RGB(i,j)** color is saved at the center and the eight link costs, as described in the Cost Function section, are saved in the 8 corresponding neighbor pixels.  The link costs shown are the average of the costs over the RGB channels, as described above (NOT the per-channel costs).  The Cost Graph may be viewed as an RGB image in the interface, (dark = low cost, light = high cost).

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | Cost(link3) | Cost(link2) | Cost(link1) | | Cost(link4) | **RGB(i-1,j-1)** | Cost(link0) | | Cost(link5) | Cost(link6) | Cost(link7) | | |  |  |  | | --- | --- | --- | | Cost(link3) | Cost(link2) | Cost(link1) | | Cost(link4) | **RGB(i,j-1)** | Cost(link0) | | Cost(link5) | Cost(link6) | Cost(link7) | | |  |  |  | | --- | --- | --- | | Cost(link3) | Cost(link2) | Cost(link1) | | Cost(link4) | **RGB(i+1,j-1)** | Cost(link0) | | Cost(link5) | Cost(link6) | Cost(link7) | |
| |  |  |  | | --- | --- | --- | | Cost(link3) | Cost(link2) | Cost(link1) | | Cost(link4) | **RGB(i-1,j)** | Cost(link0) | | Cost(link5) | Cost(link6) | Cost(link7) | | |  |  |  | | --- | --- | --- | | Cost(link3) | Cost(link2) | Cost(link1) | | Cost(link4) | **RGB(i,j)** | Cost(link0) | | Cost(link5) | Cost(link6) | Cost(link7) | | |  |  |  | | --- | --- | --- | | Cost(link3) | Cost(link2) | Cost(link1) | | Cost(link4) | **RGB(i+1,j)** | Cost(link0) | | Cost(link5) | Cost(link6) | Cost(link7) | |
| |  |  |  | | --- | --- | --- | | Cost(link3) | Cost(link2) | Cost(link1) | | Cost(link4) | **RGB(i-1,j+1)** | Cost(link0) | | Cost(link5) | Cost(link6) | Cost(link7) | | |  |  |  | | --- | --- | --- | | Cost(link3) | Cost(link2) | Cost(link1) | | Cost(link4) | **RGB(i,j+1)** | Cost(link0) | | Cost(link5) | Cost(link6) | Cost(link7) | | |  |  |  | | --- | --- | --- | | Cost(link3) | Cost(link2) | Cost(link1) | | Cost(link4) | **RGB(i+1,j+1)** | Cost(link0) | | Cost(link5) | Cost(link6) | Cost(link7) | |
| Pixel layout in Cost Graph (3W\*3H) | | |

**For help with the User Interface, see the Help menu in the sample program.**

**To Do**

All the required work can be done in iScissor.cpp, iScissor.h, correlation.cpp, and ImgView.cpp.

implement *InitNodeBuf*, which takes as input an image of size W by H and an allocated node buffer of the same dimensions, and initializes the column, row, and linkCost fields of each node in the buffer. *InitNodeBuf* MUST make use of the cross-correlation code that you'll be implementing next. You will also need to modify the eight cross correlation kernels defined in iScissor.h.

implement *pixel\_filter*, which takes as input an image of size W by H, a filter kernel, and a pixel position at which to compute the cross correlation.

implement *image\_filter*, which applies a filter to an entire image. You may do this by making calls to *pixel\_filter* if you wish.

implement *LiveWireDP*, which takes a node buffer and a seed position as input and computes the minimum path tree from the seed node. Be sure to take into account the parameter numExpanded that specifies how many levels of the minimum path tree should be expanded (which can be controlled in debug mode).

implement *MinimumPath*, which takes as input a node buffer and a node position and returns a list of nodes along the minimum cost path from the input node to the seed node in the buffer (the seed has a NULL predecessor).

**Handing in**

On CMS you will turn in two .zip archives (one for the [code](http://www.cs.cornell.edu/courses/cs4670/2015sp/projects/pa1/downloads/code.zip) and another one for the [report](http://www.cs.cornell.edu/courses/cs4670/2015sp/projects/pa1/downloads/webpage.zip)) and a .jpg image. The image will be the artifact you produce with your code (more instructions below). The class will vote on the best composites through a webpage. The first .zip will contain your code, the other one will have another copy of the artifact, and a report in the form of a webpage. We provide [webpage](http://www.cs.cornell.edu/courses/cs4670/2015sp/projects/pa1/downloads/webpage.zip) and [code](http://www.cs.cornell.edu/courses/cs4670/2015sp/projects/pa1/downloads/code.zip) .zip archives with a basic file structure. In order to have your project graded **you should not modify the organization of the boilerplate file structure**. Here are the components of the report archive:

* + **Webpage**
    - html/

Directory containig a �making of� web-page describing any extra credit items that you implemented. In the making-of webpage, besides the artifact, please include all your original images, all the masks you made, and a description of the process, including anything special you did.

* + - html/index.html

The entry point to your webpage. Do not change the name or location of this file.

* + - html/artifact.jpg

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 and save them to matte files, but then can use Photoshop or any other image editing program to process the resulting mattes (move, rotate, adjust colors, warp, etc.) and combine them into your composite. You should still turn in an artifact even if you don't get the program working fully, using the scissoring tool in the sample solution or in Photoshop (or other tools, such as the free GIMP image editor).

* + **Code**

Your submission must be a zip file containing a README.txt file and a single directory called src with source code directly inside. **The directory structure must adhere to this standard**. There will be a 5% penalty if your submission does not conform. To be clear, the following should be possible if your zipfile is in the current working directory:

$ unzip code.zip

$ ls

README.txt

src/

$ ls src/

BrushConfigUI.cpp ImageLib ImgView.cpp correlation.cpp

BrushConfigUI.h ImgFilterUI.cpp ImgView.h correlation.h

FltDesignUI.cpp ImgFilterUI.h Makefile iScissor.cpp

FltDesignUI.h ImgFltAux.cpp PriorityQueue.h iScissor.h

HelpPageUI.cpp ImgFltAux.h ScissorPanelUI.cpp imgflt.h

HelpPageUI.h ImgFltMain.cpp ScissorPanelUI.h

* + - README.txt is a text file containing any detailed information about how to run your executable (in case you added extra options or implemented extra credit items).
    - src/ is a directory containing the source code and executable for your submission.

**Bells and Whistles**

Here is a list of suggestions for extending the program for extra credit. You are encouraged to come up with your own extensions. We're always interested in seeing new, unanticipated ways to use this program!

[whistle]Implement handle in ImgView.cpp so that it applies the brush filter when ctrl is held and the left mouse button is pressed and moved. Additionally, one problem with the live wire is that it prefers shorter paths so will tend to cut through large object rather than wrap around them.  One way to fix this is to specify a specific region in which the path must stay.  As long as this region contains the object boundary but excludes most of the interior, the path will be forced to follow the boundary.  One way of specifying such a region is to use the thick paint brush that you will implement in *handle*. The next step is to modify *LiveWireDP* (if you haven't already) to take into account the brush region if "Brush Selection" is selected under "Scissors Range" in the "Scissors" menu. If this is done correctly, the live wire path will only appear within the brush selection. Also, modify the filter functionality so that it will only apply to a brushed region if "Brush Selection" is selected under "Filter Range" from the "Filter" menu.

[whistle]Modify the link costs to have the effect of blurring the image before calculating the derivatives. Note that this blurring should be applied only for the purpose of the derivative calculation and the original image itself should not be changed.  Try different amount of blurring and describe your observations on how this changes the results.

[whistle]Try different cost functions, for example the method described in  [*Intelligent Scissors for Image Composition*](http://www.cs.cornell.edu/courses/cs4670/2015sp/projects/pa1/project1_files/mort-sigg95.pdf)*,*and modify the user interface to allow the user to select different functions.Describe your observations on how this changes the results.

[whistle]The only point that doesn't snap to edges is the seed.  Implement a seed snapping feature, where the seed is automatically moved to the closest edge.

[whistle]Implement path cooling, as described in [*Intelligent Scissors for Image Composition*](http://www.cs.cornell.edu/courses/cs4670/2015sp/projects/pa1/project1_files/mort-sigg95.pdf)*.*

[whistle]Current code does not take into account the length of the path (i.e., the number of nodes in the path) when computing the minimum path. At times it can return a much longer path even if a shorter path of the same cost exists (try the checkerboard example). Correct this.

[bell][bell]Implement dynamic training, as described in [*Intelligent Scissors for Image Composition*](http://www.cs.cornell.edu/courses/cs4670/2015sp/projects/pa1/project1_files/mort-sigg95.pdf)*.*

[bell]Implement a live wire with *sub-pixel* precision.  You can find the position of an edge to sub-pixel precision by [fitting a curve (see Section 5.1.4)](http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.91.2859&rep=rep1&type=pdf) (e.g., a parabola) to the gradient magnitude values across an edge, and finding the maximum.  Another way (more complex but potentially better) of doing this is given in a [follow on](http://www.cs.cornell.edu/courses/cs4670/2012fa/readings/mortensen_99_intelligent-scissors.pdf) to Mortensen's scissoring paper.  It is probably easiest to first compute the standard (pixel-precision) live wire and then use one of these curve fitting techniques to refine it.

[bell][bell][bell]Implement [Digital Matting](http://grail.cs.washington.edu/projects/digital-matting/image-matting/).

[bell][bell][bell]Implement [Colorization](http://www.cs.huji.ac.il/~yweiss/Colorization/).

[bell][bell][bell][bell] Implement [GrabCut](http://research.microsoft.com/en-us/um/cambridge/projects/visionimagevideoediting/segmentation/grabcut.htm).