Intelligent Scissors

Problem Definition

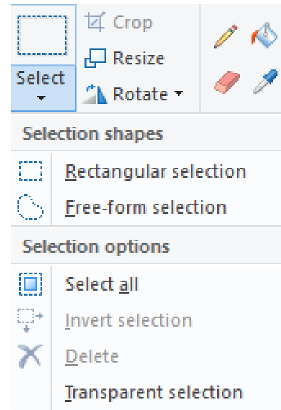

Fig1: selection in MS-Paint

Selection tools (similar to the one in MS-Paint ex: fig1) can be used to select objects in an image to resize/delete/copy/move the objects. There are many types of selection tools such as rectangles or free-form selection tool, sometimes free-form selection tools are called *Lasso's*. You can imagine a lasso as a rope surrounding your selection. Unfortunately, selection using ordinary lasso's can be tedious and boring. In Photoshop, there is a more advanced version of ordinary lasso's called *Magnetic Lasso Tool.*  Magnetic Lasso Tool is a lasso that automatically snaps to the objects' boundaries (ex: fig2). You can watch a demo of it here (local version is here).

Fig2: car selected using magnetic lasso tool

The technical term for the Magnetic Lasso Tool is *Livewire* or *Intelligent Scissors*. In this project we want to implement a simple magnetic lasso to learn more about image processing, graphs, and greedy algorithms.

Terminologies

1) **Livewire:**
    - A livewire is defined by two points and a wire (path) between them:
        i. Anchor point: a fixed point on the image the user selects at the beginning.
        ii. Free point: a moving point following the mouse cursor.
2) **Image:**
    - 2D images are usually represented as a 2D array of pixels.
    - Each pixel may contain either one (for gray images) or three (for colored images) values (fig3.a). These values are often called *Image intensity (I)*.
    - Image intensity at pixel(i,j) (I[i,j]) is the color of the image at that pixel.
    - Ex: in the rubik's cube image shown (Fig3.a), it is a 512x512 RGB image, and each pixel contains three values to represent the color RGB (Red, Green, Blue).
3) **Converting colored image to grayscale:**
    - Colored images can often be converted to grayscale by taking the average value of the image's three channel's (Red, Green, Blue) (fig3.b).

4) **Edge detection:**

- There many simple image filtering techniques (outside this course's scope) that can detect the object boundaries (edges) and tell us the position and strength of an edge at a certain pixel (fig3.c).
- An image-edge can be simply defined as a sudden change in image intensity at a certain position.
- Since these edges represent the object's boundary, we can use these edges to snap or pull the lasso towards them.

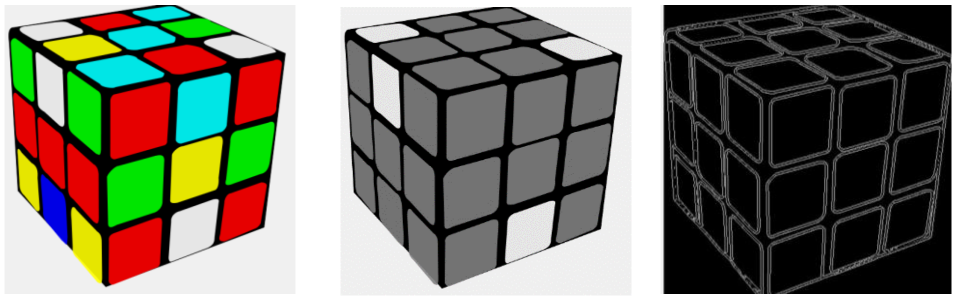

Fig3: a) colored image. b) gray-scale image. c) edge-image.

5) **Representing images by graphs:**

- Images can be represented as graphs, which can be helpful in many image analysis problems, because it reduces the problem from an image domain problem to a graph domain problem. On such graphs you can apply typical graphs algorithms (Dijkstra, BFS, DFS, …etc) to solve the problem at hand.

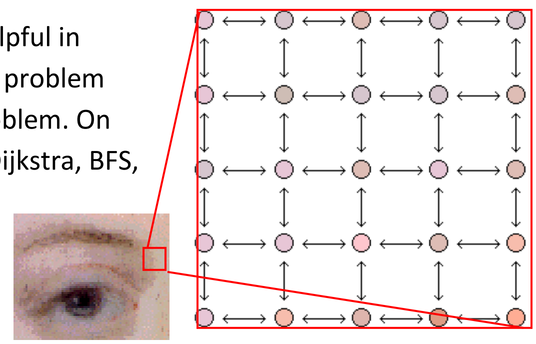

Fig4: image pixels represented as graph vertices (nodes), and the pixels' neighborhood is represented as graph edges.

6) **Graph construction:**

- To construct an undirected weighted-graph we need to define:
    i. Vertices (nodes).
    ii. Connectivity (edges).
- This image-graph can be structured as follows:
    i. Vertices: Each image pixel is considered as a vertex in the graph. So if we have an NxM image then we have an NxM vertices in our graph.
    ii. Connectivity: there are many ways to connect the vertices grid, the simplest way is to establish a 4-connectivity (fig5). So we need to connect each pixel with the pixel on the above, below, left, and right.

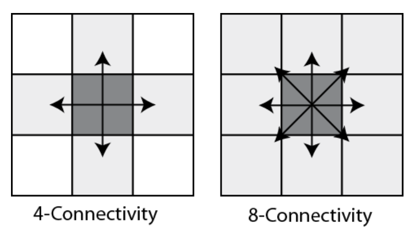

Fig5: connectivity types

7) **Mapping a "livewire in an image" problem to a "shortest path in a graph" problem:**

- Assuming that we have an undirected weighted-graph for the image, with **small weights** on the objects' boundaries (**image-edges**) and **large weights** at the **homogenous parts** of the image.

- If we need to generate a livewire between two pixels P1(i,j) and P2(x,y), it is the same as getting the shortest path between the two corresponding vertices V1(i,j) and V2(x,y), because the low edge-weights are at the image-edges on which we want our livewire to snap on.
- Now remains one issue towards constructing our graph, which is determining the edges' weights between pixels.

8) **Edge Weights Generation:**
   - Assuming that you have a value G that measures the image-edge strength and direction between two pixels P1 and P2 (Since, calculating G is out of the course's scope. You will be supplied with a function that calculates G between two given pixels).
   - Then we can set the edge-weight between P1 and P2 as $W_{p1,p2} = 1/G$. so regions with Low G have high weight, and regions with high G have low weight.

Project Requirements

Provided Implementation

1. Template for opening and displaying the images.
2. Function to calculate the edge-strength G between two pixels.

Refer to Appendix: Template Code Description for more details.

Required Implementation

Requirement	Performance
1. Construct an undirected weighted-graph for a given image.	**Time: should be bounded by $O(N^2)$** **N: width/height of the image**
2. Calculate the shortest path EFFICIENTLY from an anchor pixel (vertex) to all pixels (vertices in the graph).	**Time: should be bounded by O(E` lg(V`))** V`: # of vertices that are checked until reaching the destination E`: # of edges that are checked until reaching the destination, E` = $O(N^2)$
3. Backtrack the shortest path from a free point (mouse position) to the anchor point.	**Time: should be bounded by O(N)**
4. Draw the path on the image.	**Time: should be bounded by O(N)**
5. Generate a sequence of connected paths using multiple anchor points.	**Shortest path calculation: should be bounded by $O(N^2 \lg(N))$** **Shortest path drawing: should be bounded by O(N)**
6. When the user finish selection close the lasso by generating a path between the last and first anchors.	**Time: should be bounded by O(N)**

Input

1. Image (2D array of pixels).
2. Anchor point.
3. Free point.

Output

1. Path between the anchor and the free point.

2. Final lasso closed path.

Test Cases

- The algorithm can be tested on any picture on the computer.

Deliverables

Implementation (60%)

1. Graph construction.
2. Shortest path EFFICIENT implementation.
3. Path backtracking and drawing.
4. Support multiple anchor points.
5. Generating a closed lasso.

Document (40%)

1. Graph construction description and code.
2. Used shortest path algorithm code.
3. Detailed analysis of the above codes.

Allowed Codes

- Given template to:
    1. Open and display the images.
    2. Function to calculate the edge-strength G between two pixels.

    Refer to Appendix: Template Code Description for more details.
- No other external code is allowed.

Milestones

<table>
<tr><th></th><th>Deliverables</th><th>Due to</th></tr>
<tr><td>Milestone1</td><td>1. Construct a weighted graph for an image
2. EFFICIENT implementation of shortest path between start anchor point and free points.
3. Documentation I</td><td>START of week 13
[Week before LAB EXAM]</td></tr>
<tr><td>Milestone3</td><td>1. Backtrack the shortest path from a free point (mouse position) to the anchor point.
2. Draw the path on the image by supporting multiple anchor points.
3. Documentation II</td><td>END of week 14
[LAB EXAMS WEEK]</td></tr>
<tr><td colspan="3">For Milestone1:
o **MUST** deliver the required tasks and **ENSURE** it’s worked correctly
o **MUST** deliver the **part of the documentation** that is related to the Milestone (printed document)
o **MUST** deliver in your scheduled time (TO BE ANNOUNCED)</td></tr>
</table>

BONUSES

1. As you can see in Photoshop example you can 1) Click to place anchor. 2) Move the mouse to generate the livewire. 3) When the wire's length exceeds a certain length, an automatic anchor point is placed to make the wire more stable.

   Bonus: implement a similar algorithm that automatically places new anchor points.
2. Add the ability to increase the frequency of anchor points in some critical regions (or any other Photoshop-like features as shown here)
3. Faster implementation for the shortest path to be less than the given bounded complexity above, i.e. to be less than **O(E` log V`).**

Appendix: Template Code Description

C# Code contains **ImageOperations** class with the following functionalities:

1. Open image & load it in a 2D array stored in a global variable of type `MyColor`[1] `[,]` called `ImageMatrix`

   ```
   MyColor [,] OpenImage(string ImagePath)
   ```

2. Get width and height of the image matrix

   ```
   int GetHeight(MyColor[,] ImageMatrix)
   int GetWidth(MyColor[,] ImageMatrix)
   ```

3. Calculate the energy between two pixels

   ```
   double CalculatePixelsEnergy(MyColor Pixel1,MyColor Pixel2)
   ```

4. Display an image on a given `PictureBox` control

   ```
   void DisplayImage(MyColor[,] ImageMatrix,PictureBox PicBox)
   ```

[1] `MyColor` is a structure defined in the code to hold the Red, Green, Blue values of each pixel