Intelligent Scissors

[Problem Definition 2](#_Toc435829281)

[Terminologies 2](#_Toc435829282)

[Project Requirements 4](#_Toc435829283)

[Provided Implementation 4](#_Toc435829284)

[Required Implementation 4](#_Toc435829285)

[Input 4](#_Toc435829286)

[Output 4](#_Toc435829287)

[Test Cases 5](#_Toc435829288)

[Deliverables 5](#_Toc435829289)

[Implementation (60%) 5](#_Toc435829290)

[Document (40%) 5](#_Toc435829291)

[Allowed Codes 5](#_Toc435829292)

[Milestones 5](#_Toc435829293)

[BONUSES 5](#_Toc435829294)

[Appendix: Template Code Description 7](#_Toc435829295)

# Problem Definition

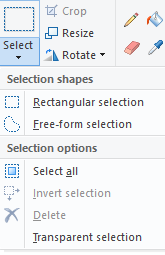
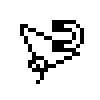
Selection tools (similar to the one in MS-Paint ex: fig1) can be used to select objects in an image to resize/delete/copy/move the objects. There are many types of selection tools such as rectangles or free-form selection tool, sometimes free-form selection tools are called *Lasso’s*. You can imagine 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](https://www.youtube.com/watch?v=0rQEctxkjBM) ([local version is here](../Demos/Intelligent%20Scissors/Photoshop%20How%20to%20use%20the%20Magnetic%20Lasso%20Tool.mp4)).

Fig1: selection in MS-Paint

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:
     1. Anchor point: a fixed point on the image the user selects at the beginning.
     2. 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.

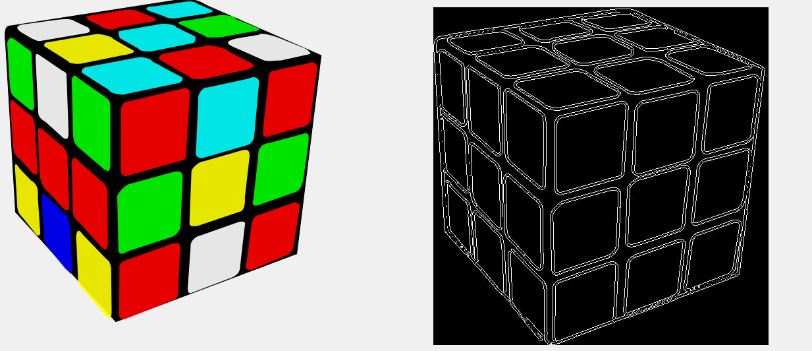
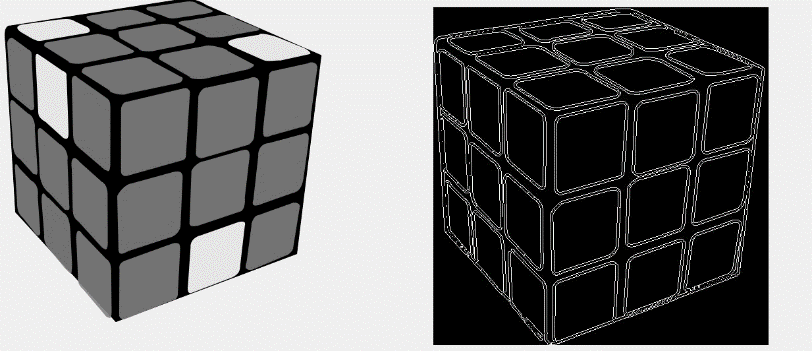
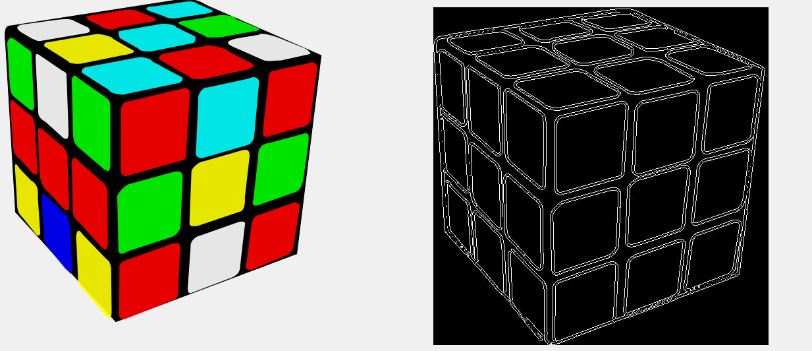


Fig3: a) colored image. b) gray-scale image. c) edge-image.

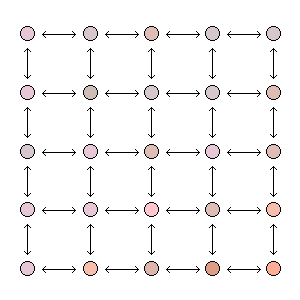
1. **Representing images by graphs:**
   * **thumb_mot.gif**Images can be represented as graphs, which can be helpful in many image analysis problems, because it reduces the problem from an image domain problem to a graph domain problem. On such graphs you can apply typical graphs algorithms (Dijkstra, BFS, DFS, …etc) to solve the problem at hand.
2. **Graph construction:**
   * To construct an undirected weighted-graph we need to define:
     1. Vertices (nodes).

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

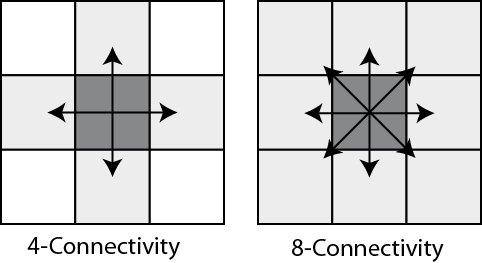
* + 1. Connectivity (edges).
  + This image-graph can be structured as follows:
    1. Vertices: Each image pixel is considered as a vertex in the graph. So if we have an NxM image then we have an NxM vertices in our graph.
    2. Connectivity: there are many ways to connect the vertices grid, the simplest 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

1. **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.
2. **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 Wp1,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 for more details.

## Required Implementation

|  |  |
| --- | --- |
| **Requirement** | **Performance** |
| 1. Construct an undirected weighted-graph for a given image. | **Time: should be bounded by O(N2)**  **N: width/height of the image** |
| 1. 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(**N2**) |
| 1. Backtrack the shortest path from a free point (mouse position) to the anchor point. | **Time: should be bounded by O(N)** |
| 1. Draw the path on the image. | **Time: should be bounded by O(N)** |
| 1. Generate a sequence of connected paths using multiple anchor points. | **Shortest path calculation: should be bounded by O(N2 lg(N))**  **Shortest path drawing: should be bounded by O(N)** |
| 1. 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

* Sample Cases
* Complete Cases

# 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 for more details.

* No other external code is allowed.

# Milestones

|  |  |  |
| --- | --- | --- |
|  | **Deliverables** | **Due to** |
| **Milestone1** | 1. Construct a weighted graph for an image 2. EFFICIENT implementation of shortest path between start anchor point and free points. 3. Backtrack the shortest path from a free point (mouse position) to the anchor point. 4. Draw the path on the image by supporting multiple anchor points. 5. Documentation | **(START of Week Before Practical Exam)** |
| **For Milestone1:**   * + **MUST** deliver the required tasks and **ENSURE** it’s worked correctly   + **MUST** deliver in your scheduled time (TO BE ANNOUNCED) | | |

# BONUSES

1. As you can see in [Photoshop example](https://www.youtube.com/watch?v=n6nHAnc0E2E&feature=youtu.be) 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.

1. Add the ability to increase the frequency of anchor points in some critical regions (or any other Photoshop-like features as [shown here](https://www.youtube.com/watch?v=0rQEctxkjBM))
2. 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]](#footnote-2) [,]called ImageMatrix

MyColor [,] OpenImage(string ImagePath)

1. Get width and height of the image matrix

int GetHeight(MyColor[,] ImageMatrix)

int GetWidth(MyColor[,] ImageMatrix)

1. Calculate the energy between two pixels

double CalculatePixelsEnergy(MyColor Pixel1,MyColor Pixel2)

1. 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 [↑](#footnote-ref-2)