Intelligent Scissors

# TA: Mohamed Raafat

## T093

|  |  |
| --- | --- |
| منة الله إبراهيم محمد عفيفي  أحمد محمد توفيق محمد  أيمن مصطفى حسن حسين | 20191700665  20191700062  20191700162 |

Abstract

Selection tools 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 is a tool that automatically snaps to an Objects’ Boundaries.

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.



Contents

* [Graph Construction](#_Graph_Construction)
* Calculating Shortest Path
* Backtracking Shortest Path
* Frequency
* Scaling small Images
* Cropping Final Image

# Graph Construction

• Initializing *adj* List<> -> ->

## private static void constructGraph(RGBPixel[,] ImageMatrix)

• Loops on pixels to add the weights of neighboring pixels

• Main Complexity:

• Two nested loops looping on the pixels of the Image -> ->

• Helpers:

• addEnergyFor(int row, int col, Vector2D pixelEnergies) ->

• addEnergyToNeighbour(..) ->

Note: List<>.Add() is considered O(1) since the initial capacity of the List after adding the first element is **4** and the maximum edges per vertex is **4** so the List<> won’t need the O(N) expansion

• Final Complexity:

• Notes:

Calculating Shortest Path

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.

A picture containing text

Description automatically generatedTo get the shortest path between two vertices in an undirected weighted graph we use Dijkstra Algorithm.

Analysis of Dijkstra

How fast is Dijkstra? It uses a Priority Queue to keep track of the smallest Cost, we use a Fibonacci Heap to implement the priority queue needed; which improves the asymptotic running time.

Fibonacci Heaps have a better amortized running time than many other priority queue Implementations including binary heaps.

The main Fibonacci Heap Operations used are:

* Enqueue (Insert in Fibonacci heap) -> which takes constant time O(1)
* Dequeue (Extract Min) -> which works in O(log n) amortized time

Dequeue is used in the outer while loop (Line 47) which iterates for every vertex in the graph,

Enqueue is used in the inner for loop (Line 61) which visits all edges for every vertex.

Therefore, Dijkstra has a complexity of O( V’ log V’ + E’) using Fibonacci Heaps,

Where V’ and E’ are number of vertices and edges checked until reaching the Destination.

Backtracking the Shortest path

After using Dijkstra to get the shortest path from the last anchor to a free point, we need to

Construct the path by backtracking from the destination node to the source node, We get to

Each node’s parent till we find a node that has no parent, this is the source node.

Graphical user interface, text, application

Description automatically generated

Analysis

The while loop starts with the destination and ends when we have found the source,

Therefore, it has a complexity of O(N) where N is the length of the shortest path.