CS21120 Assignment 2 (25% of module mark)

Image contour segmentation

Release date Friday 27th February 2015
Hand in date Monday 13th April 2015 (23:59 via Blackboard)
Feedback date Monday 4th May 2015 (via Blackboard)

Aims and Objectives

The aim of this assignment is to give you experience in writing graph algorithms and using some basic data structures. The objective is to write a system for assisting with extracting edges in images based on Dijkstra's shortest path algorithm.

Overview

Strong edges in images (i.e. transitions from light to dark regions) are often important for applications in image processing and computer vision. In this assignment you will be completing an implementation of an interactive edge segmentation algorithm based on Dijkstra's shortest path algorithm on graphs. The user selects a seed point and, as they move the mouse around, the lowest cost path back to the seed is selected and displayed to the user.

The algorithm works as for the normal Dijkstra's algorithm, treating each pixel as a node in a graph and using several filtered versions of the image as the edge weights between neighbouring pixels. The filtered images are provided for you.

Graph representation

In this assignment each pixel of the input image should be treated as a node in the graph and each pixel should be connected to its 8 neighbours as shown below in Figure 1. Due to this very regular structure of the graph it makes sense to use images (i.e. 2D arrays of floats) to store the edge weights. The graph is undirected, so only 4 images are required for pixel (x,y): North (to (x, y+1)), East (to (x+1, y)), North-East (to (x+1, y+1)) and North-West (to (x-1, y+1)). The other 4 edges are given by:

- South(x,y)=North(x,y-1),
- SouthEast(x,y)=NorthWest(x+1,y-1),
- West(x,y)=East(x-1,y)
- SouthWest(x,y)=NorthEast(x-1,y-1).

This is illustrated in Figure 2. The four edge images are passed to the *setSeed* method in the supplied *ISnapper* interface that you will need to implement.

Requirements

You need to implement Dijkstra's shortest path algorithm for image edge extraction. Outline code is provided that handles loading and display of images and display of the extracted lines. You only need to supply an implementation of the *ISnapper* interface that performs the edge extraction. This requires the implementation of the *setSeed* method and the *getPath* method. The *setSeed* method is called when the user presses the mouse down on the image and should start the process of building the "map". The map should indicate for each pixel which neighbour is next on the shortest path

back to the seed point. The *getPath* method is called as the user drags the mouse around, and should use the calculated map to return the shortest path back to the seed point.

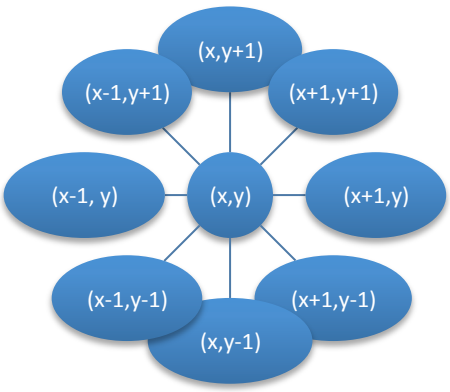

Figure 1. Showing the connectivity of a single pixel at location (x,y) to its 8 neighbours.

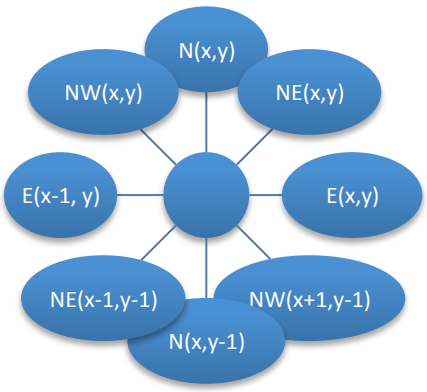

Figure 2. Showing the storage of edges for pixel (x,y) in the 4 images N, E, NE and NW.

Suggested implementation

Dijkstra's algorithm requires three main additional pieces of information during its execution: (1) a set of edges, from which the lowest total cost edges are selected as the map is built, (2) a visited flag for each node (i.e. pixel) and (3) the map indicating for each pixel which neighbour is next on the shortest path back to the seed point. The edges (1) are best stored in a priority queue (use the thread safe *java.util.PriorityBlockingQueue*) for efficient selection of the best edge to use next as the map is built. You will need to create a class representing a single edge i.e. start point, end point and weight, which can be stored in the priority queue. This class should implement the *Comparable* interface to allow correct sorting by the priority queue. To allow user interaction while the map is building, it is recommended to do the map building in a separate thread. The visited flags (2) are best stored in a 2D array of *boolean*, the same dimensions as the image. The map (3) can be stored in a 2D array of *java.awt.Point*.

Marking

The following is a rough guide to the breakdown of marks based on the structure of my code. If your code is structured differently that is OK, provided it works.

1) *Class structure* (20%): A sensible structure for the main *ISnapper* implementation and the support class for weighted edges.
2) *setSeed initialisation* (20%): The *setSeed* method of your *ISnapper* needs to initialise the data required to build the map, add the seed points edges to the priority queue and start the map building thread.
3) *Map building* (20%): The map building should take the next edge, $\boldsymbol{p}$, off the *PriorityBlockingQueue*, and (if $\boldsymbol{p}$*.target* hasn't been visited):
    a. Set the target point's visited flag to *true* e.g. *visited[p.target.x][p.target.y]=true*
    b. Set the map value at the target point of the edge equal to the edge's start point e.g. *map[p.target.x][p.target.y]=p.start*.
    c. Add the target point's edges $\boldsymbol{q}_i$ to the *PriorityBlockingQueue* with weight = $\boldsymbol{p}$*.weight* + $\boldsymbol{q}_i$*.weight.*

This is continued until the *PriorityBlockingQueue* is empty.

4) *getPath method* (20%): The *getPath* method should use the map to calculate the path from the specified pixel to the seed point location. If the selected location is outside of the image bounds or outside of the calculated part of the map it should return *null*. Otherwise it should iteratively add the point at the current location to the *LinkedList* of points, then step to the point specified at this location and continue until reaching a *null* (which should be the seed point).
5) *Javadoc formatted documentation* (20%): Instead of a separate report you are asked to thoroughly document your source code using *javadoc* comments. This should include a comment at the top of the main class explaining what was achieved and any problems faced. Each method should include a properly formatted javadoc comment including a comment and any @param and @return tags with a comment for each. Do not just use the @Override annotation for the implemented methods (*setSeed* and *getPath*) as I want a description of your implementation and any problems faced.

Submission

For this assignment you are asked to submit a **single** *.java* file of source code via Blackboard, which ***must*** be named *<uid>Snapper.java*, where *<uid>* should be replaced with your user ID. For example mine would be called *BptSnapper.java*. This file should include your *ISnapper* implementation and any support classes (these could be inner or just private), such as the weighted edge *Comparable* class. Your solution ***must*** be defined in the package *cs21120.assignment2.solution*. This is for ease of marking, not for good design considerations.

Resources

You are provided with outline classes for image loading and filtering as a compiled .jar file and accompanying *javadoc* documentation. You should only need to implement the *ISnapper* interface. The main program (in the jar file) is called *LineSnapper*. You can supply your class to *LineSnapper* either by selecting "Set Snapper" from the File menu and using the GUI provided, or by passing the full name of your class (including full package details) as a command line parameter. The only other class you should need to interact with directly is *FloatImage*, as a set of *FloatImage*s are passed to *setSeed* to represent the edge weights. You should only need to use the *getWidth*, *getHeight* and *get_nocheck* methods of *FloatImage* in your program.

You may use material from the lectures / course, but it is your responsibility to check any material used and correct if needed (but please report any bugs you find). Any resources you use (including these) must be acknowledged in your javadoc comments.

Academic conduct

As with all such assignments, the work must be your own. Do not share code with your colleagues and ensure that your own code is securely stored and not accessible by your classmates. Any sources you use should be properly credited with a citation, and any help you get (e.g. from demonstrators) should be acknowledged. Your comments must accurately reflect what you have achieved with your code, any discrepancies between your code and comments could be treated as academic fraud.

Bernie Tiddeman - 27th February 2015