Long Assignment: Dijkstra's algorithm

Due date: in class Thursday, Nov 2.

Assignment: Implement two variants of Dijkstra's algorithm and compare their performance. Write up your findings in 2500-3000 words. Details follow.

Feel free to consult Wikipedia, which has a decent description of Dijkstra's algorithm with conventional priority queue.

***

In this mini-project, you will investigate the performance of best-first single-source shortest path algorithms in the context of geodesic distances. Best-first shortest path algorithms are exemplified by Dijkstra's algorithm. Here, geodesic distances are distances over the image plane where the distance between adjacent pixels is some fixed horizontal distance plus a vertical distance given by the color difference between the pixels, combined using Pythagoras.

Best-first shortest path planning algorithms work as follows. You begin with a graph in which every distance is unknown except for a set of start nodes, all with distance zero. The start nodes are put into a priority queue. At every step, the top node, say p, is removed from the queue, p's distance is finalized, and then its neighbours are examined. For every neighbour n, you compute a hypothetical distance: the distance of p plus the distance between p and n. If the hypothetical distance for n is less than its previously stored distance -- or n's distance was previously unknown -- node n is updated and added to the priority queue. Eventually, all nodes have their distances finalized, and the algorithm terminates.

We will look at this algorithm for graphs over images, where each pixel is connected to its four immediate neighbours, up, down, left, right. In geodesic applications, the incremental distance between p and n is sqrt(1+g*colordist(p,n)) where g is a constant and colordist(.) is a function measuring the color difference between the pixels given as arguments.

The algorithm described above is fairly straightforward. The main complexity lies in managing the priority queue; feel free to use a library for that. You will investigate two variants of the method. First, one where the priority queue uses the conventional "decrease-key" operation to update node values. Second, a version where nodes are not updated in the queue, but rather in the graph, and an updated node is inserted into the queue as a duplicate.

Nodes in the queue can therefore become stale, and you need to check whether a node is up to date (does its graph distance agree with the priority queue distance?) before processing it. Stale queue entries can safely be discarded.

One you have the implementation of both variants, compare them. Confirm their correctness by making sure they give the same values (and maybe check with a friend).

Which is faster? Run them over a few dozen or a few hundred images to collect data. Then, write up your experience with the algorithms in 2500-3000 words (roughly 6 pages). Consider some of the following questions:

1. The theoretical runtime of the algorithm is O(N log N) for N nodes in the graph, i.e., N pixels. Run the method with various image sizes. Can you confirm the upper bound?

2. In the case of a heap-based priority queue, the runtime is actually O(N log K) for K nodes in the heap. What is the largest heap size you saw in each variant? Here "largest" should be as a fraction of the image pixels --

you can always get a bigger heap in absolute terms by running on a bigger image.

3. What do the shortest paths look like? Create a few visualizations of the distances on different input images. Two suggestions are (1) show the distance as a greyscale value over the image plane; (2) show several shortest paths overlaid on the original image.

4. Does your choice of starting node affect the runtime? For example, is there a difference between picking the centre, the upper left corner, or the lower right corner?

5. Select a starting node in the top left and compute the distance to the lower right: this distance provides an estimate of the "diameter" of the image. Compute the diameter for several images of the same size. How much variability is there? What kinds of images have larger diameter and which produce smaller values? Why?

6. What happens when you have several starting points distributed over the image plane? Obviously the distances will be completely different, but what happens to the runtime?

7. The original algorithm was designed for arbitrary graphs, but images have a highly regular structure. Can you think of ways to speed up the algorithm by exploiting its structure?

8. You don't need to limit yourself to these questions. If you have additional experiments or observations about the algorithm, you can report them.

Last modified: Wednesday, 4 October 2017, 10:58 AM