Flickrbooks

Ronit Slyper

December 16, 2007

1 Introduction

Flickrbooks tested whether the Flickr dataset contained enough images to span a space wherein any two images could be connected by a perceptually smooth path of images. The result is a very definite "probably" [1].

2 Working with lots and lots of data

The dataset I worked with was */nfs/baikal/jhhays/flickr_geo_and_gps*. I had foolishly decided to use my own distance metrics rather than James's. I thus used the 2.6 million images from this dataset for which 64x64 and 8x8 thumbnails had been / could be created in the time alloted.

I calculated 8x8 as the maximum size wherein all images could fit into memory at once. I created a $400MB$ flatfile of these images.

To find the closest neighboring image to a particular image, the following algorithm was run.

```
A = image
Heap H
threshold = constant
for all images e ∈ 8x8 do
  if dist8(e, A) < threshold then
    H.insert(e)
  end if
end for
top = constant {constant = around 50}
closest = ∅
for i = 1 to top do
  h = H.pop_minimum()
  load h's 64x64 image into memory
  if dist64(h, A) is the smallest so far then
    closest = h
```

[1] Given over two million images, full use of a Yahoo cluster, more than a week and a half, an incontestable measure of "perceptual continuity", James's image descriptors, and Jim to stop saying "I told you so" about everything I try.

16: **end if**
17: **end for**
18: **return** *closest*

For the graph algorithms described below, I defined graph connectivity as "all images are connected to their 20 nearest neighbors".

3 Source-Sink Paths

The project was begun with the goal of being able to find a path between two pre-selected images. I considered all-pairs shortest path, Dijkstra's shortest path algorithm, $A^\star$, and beam search for this task. All-pairs rapidly proved infeasible[2]; beam search is effectively being automatically implemented by the arbitrary restriction of graph connectivity.

3.1 Dijkstra's

Dijkstra's was the first reasonable search algorithm implemented. The entire graph structure was precalculated once and stored to disk. Paths between two images could then be calculated swiftly using Fibonacci heaps in $O(|V|log|V|)$. Dijkstra's is not, however, optimal for this application, as it is a breadth-first search algorithm. The problem was most obvious when the two images given were not connected in the graph; Dijkstra's spent a copious amount of time searching an entire half of the graph before returning "disconnected".

3.2 $A^\star$

The depth-first search algorithm $A^\star$ attempted to rectify Dijkstra's long-winded meanderings. $A^\star$ uses a distance heuristic to nudge the path in the right direction; a natural heuristic for this application was image distance using the distance function already in use.

$A^\star$ is especially useful in situations where the graph structure is not known in advance. This is because $A^\star$ does not require the full structure, desiring only expansion of promising nodes. All graph connectivity computation was thus done on-line, on-demand. This led it to be considerably slower than Dijkstra's; never, in fact, returning a path! It is also possible that the heuristic used, being computed on thumbnails and thumbnails of thumbnails, was not entirely admissible, resulting in portions of exponential growth.

4 Random Paths

Although $A^\star$ has been implemented (and works, should we ever desire to compute a path between two specific images), for the purpose of this project it

[2] "I told you so," quoth the TA

was unnecessary to constrain the problem to a given starting and ending image. Random paths can be computed in $O(|V|p)$, where p is the path length. In practice, each additional path segment requires only 1 to 3 seconds. No graph structure needs to be computed or precomputed. The following simple algorithm, using the previously-defined closest-neighbor algorithm, was run.

1: v_0 = random node
2: SET S
3: $S.insert(v_0)$
4: **for** $i = 1$ to $p - 1$ **do**
5: $\quad v_i$ = closest image not in S
6: $\quad S.insert(v_i)$
7: **end for**
8: **return** $\{v_0, v_1, \ldots, v_{p-1}\}$

5 Distance Metric

The distance metric was the clear weak point of this project. Several distance metrics for comparing images were tried:

- Pixel distance
- Normalized pixel distance (subtract the average μ, divide by σ before comparing)
- Pixel distance, with a penalty for being a grayish image
- Gradient stuff

The first two yielded equivalent results; the third results in bright orange images. I also tried simple gradient comparisons, but was limited by time constraints from pursuing this further.

The problem with color is that the images in paths tend to become gray and boring; a gray image is a better match for any image than something more exciting. I was unable to overcome this problem.

5.1 An Interesting Note on Aspect Ratio

Roughly half the pictures in the flickr dataset were at an aspect ratio of 640x480.

I began my project using the full two million images. When comparing images in the 8x8 space or 64x64 space, the images need to span the full square size in order to be comparable. Thus I duplicated the last row/column of the thumbnail until it was square. Results were good, but I suspected that they might be better if I wasn't giving such unfair weight to that one row/column. Thus I switched my code over to using only the images with the 640x480 aspect ratio (actually, to those images with an 8x6 thumbnail, which, to my chagrin, I later realized didn't necessarily correspond to a 640x480 ratio, resulting in some ugly hacks; but I digress). The results were *significantly worse*. Thus I believe

that at least two million images are needed to span the space of images, and I also believe that more images would *significantly improve* the results; it would be interesting to determine the exact cutoff where the space is spanned.

6 Visualization

6.1 Movies

With such quick path computation, the limiting computation factor in this project was, in fact, rendering the movies[3]! I displayed the images as blended textures using OpenGL, called glReadPixels to grab the frame buffer, and saved the images to disk. Images sequences were assembled using mencoder.

I also created a simple viewer, with blending, to view the paths before deciding to render them. In this way I could make use of having lots of data at my disposal: create and view tons of paths, and only choose the best!

Image blending was controlled using a linear function plus a sine wave.

6.2 Motion

Morphing between two images was implemented by splatting a uniform grid onto the first image, then adjusting each of the image patches at the gridpoints until they best matched the gradients of the image patches in the second image. The patches were animated using the blend function described above. Results were trippy.

6.3 Graph Visualization

I remain intensely curious about the graph structure created by the 20-nearest-neighbors criterion. I attempted to visualize it using graphviz, writing a program to output my connectivity graph as a .dot file. The result swamped the program and also crashed xpdf. It would be interesting to make further attempts at this in the future. How many components does the graph have? Is it clustery in source space, or uniform? Does it have many outliers, a hub structure, ... ?

7 Conclusion

This project was a fascinating introduction to the techniques required to deal with lots of data. The results created along the course of my experiments were sometimes cool, often funny - bagpipers morphing into sheep, a white-sky/blue-waves morphing into white-snow/blue-sky, an entire movie concocted of vertical horizon lines from images people neglected to rotate - and always jolly surreal. Flipbooks have made it into the information age - in the form of Flickrbooks!

[3]Technically, the limiting reagent was disk speed for saving the images.