CSCI-1200 Data Structures — Fall 2009
Homework 9 — Perfect Hashing

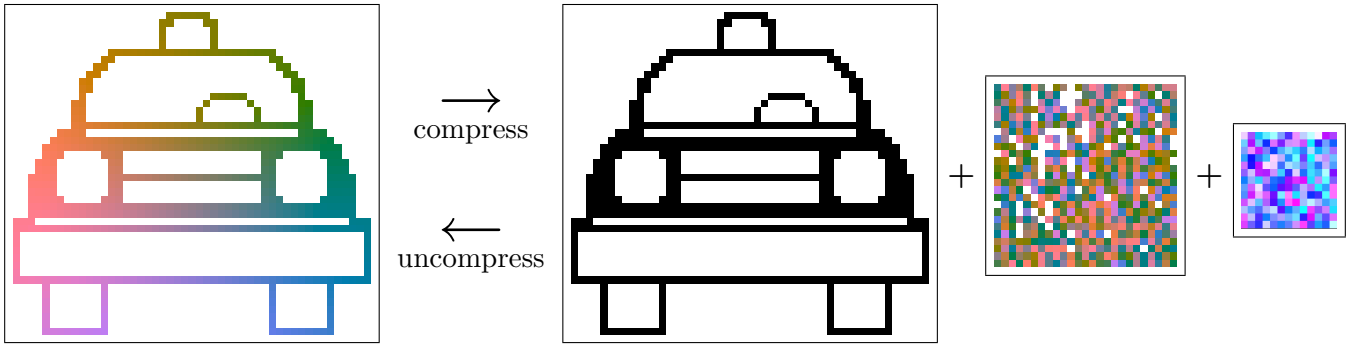
For this assignment you will implement a form of lossless image compression based on perfect hashing. A *perfect hash function* for a given set of keys maps those keys to the hash table with *no collisions*. This assignment is based on a recent paper from SIGGRAPH, a prestigious academic computer graphics conference:

"Perfect Spatial Hashing". S. Lefebvre, H. Hoppe. SIGGRAPH 2006, pages 579-588.
http://research.microsoft.com/~hoppe/perfecthash.pdf

You are encouraged to read this paper — although it is not necessary to understand all of the details to complete the assignment. We will only be implementing a small portion of the system described in the paper. *Be sure to read the entire handout before beginning your implementation.*

Using Hashing to Compress Images

The input to this image compression scheme is a 24-bit-per-pixel image which has many, many pure white pixels (e.g., the leftmost image below). In other words, the non-white pixels are sparse. This compression scheme *will not* be effective on typical real-world photographs, where the lighting and shading gradients mean that the number of pixels with exactly the same color (*any* color) is quite small.



input	**occupancy**	**hash data**	**offset**
49x44 pixels	49x44 pixels	25x25 pixels	13x13 pixels
24 bits/pixel	1 bit/pixel	24 bits/pixel	8 bits/pixel
6481 bytes	317 bytes	1888 bytes	185 bytes

The output of the compression consists of 3 files. First, a 1-bit-per-pixel *occupancy* image, which is black (true) where the input image contains data (the non-white pixels), and white (false) everywhere else. Second, a densely packed *hash table* containing the non-white pixel data is stored as a 24-bit-per-pixel image, we call the hash data image. The size of this table depends on the number of non-white pixels in the original image. Finally, a small 8-bit-per-pixel image is used to store the *offset table*, which represents the hash function. In the above example, the total compressed representation is 2390 bytes, which is a 73% compression from the original representation.

Because the hash function is precisely constructed so that there are *no collisions*, we do not need to implement separate chaining or open addressing to resolve collisions. Furthermore, the hash table only needs to store the value (the pixel color) and *does not* need to store the key (the original pixel coordinates). Importantly, this representation allows random access to any pixel *without* uncompressing the entire image!

Using the Hash Function

Your first task for this assignment is to implement the uncompression phase. This basically boils down to copying the occupancy image to a full color 24-bit-per-pixel image, and replacing all black pixels with the color data from the appropriate pixel in the hash data image.

So how do we lookup a pixel (i,j) in the hash table? First we grab the offset value, **offset**(i % s_{offset}, j % s_{offset}), where s_{offset} is the size of the offset image. The offset value is split into dx, a 4-bit offset value in the x direction, and dy, a 4-bit offset value in the y direction. Then we can calculate the corresponding location in the hash data image, **hash data**($(i+dx)$ % s_{hash}, $(j+dy)$ % s_{hash}), where s_{hash} is the size of the hash data image. Here's an example command line to perform uncompression:

```
a.exe uncompress occupancy.pbm data.ppm offset.offset output.ppm
```

In addition to visually inspecting the results, you may check that your program correctly and losslessly uncompressed the data with this command:

```
a.exe compare input1.ppm input2.ppm output.pbm
```

which prints the number of non-matching pixels and creates a 1-bit-per-pixel *difference image* that is black (true) where the pixels are the same, and white (false) where the pixel colors are different.

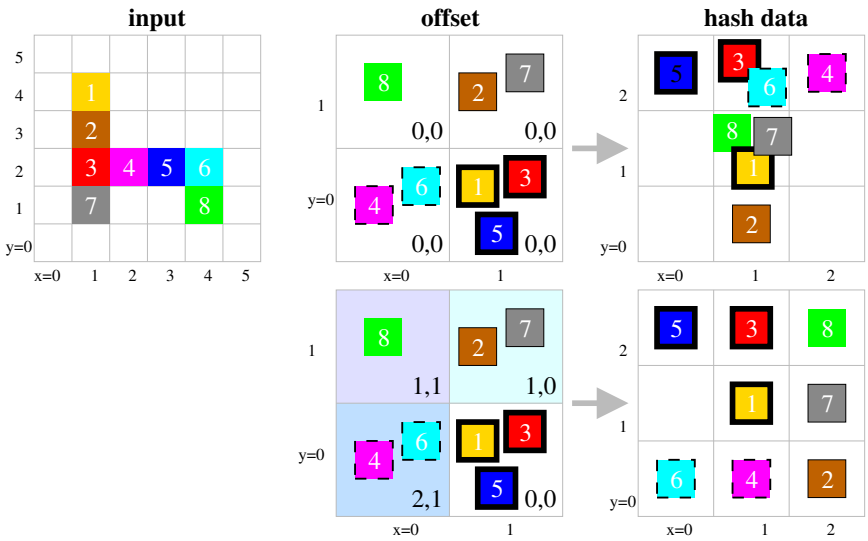
Constructing the Hash Function

The second part of the assignment is to perform image compression. The real challenge, of course, is to construct a small, perfect hash function. To guarantee that there are no collisions, we must be given all of the keys (non-white pixel coordinates) ahead of time. After the function is constructed, the user is allowed to change the color of the non-white (data) pixels, but *cannot* add new data pixels.

Rather than representing the hash function as a mathematical formula or a typical C++ function, the hash function is stored as an offset image. What's a good size for the offset image? What's a good size for the hash data image? Lefebvre & Hoppe recommend starting with:

$$s_{hash} = \left\lceil \sqrt{1.01 * p} \right\rceil \qquad\qquad s_{offset} = \left\lceil \sqrt{\frac{p}{4}} \right\rceil$$

where p is the number of non-white pixels in the input image. In the example below we have a 6x6 input image with $p = 8$ non-white pixels. Thus, we select $s_{hash} = 3$ and $s_{offset} = 2$. First, let's consider what happens if we assign all offsets to be 0. As shown in the top row below, we will get collisions in two of the hash data cells.



Example from: http://research.microsoft.com/~hoppe/perfecthash_sig2006.ppt

How do we assign offset values to separate these collisions? The idea is to first assign the offset values that are used a lot, since they will control the placement of many values and will be the trickiest to separate

from each other. We sort the offset cells by the number of values mapping to each offset cell: {1,3,5}, {2,7}, {4,6}, and {8}. We first choose an offset for the cell containing 1, 3, and 5. Since nothing has been assigned to the hash table, anything will do. We choose (0,0). Then, we take the next most populous offset table cell, containing 2 and 7. We search through all possible offsets to find one that does not collide with the values already in the table. (1,0) works. Etc. As we work our way through the sorted list, the hash table will have fewer empty cells making it harder to choose a collision-free offset. But fortunately, the number of entries we are placing is also decreasing so it is still quite likely we will find a spot.

We are *not* guaranteed to find an assignment of offset values that completely separates the data with this *greedy* method. To guarantee that we find an appropriate assignment of offset values, *if it exists*, it would be necessary to implement a search over all assignments (you may do this for extra credit). If the greedy method fails to find a perfect hash function for a particular choice of s_{hash} and s_{offset}, Lefebvre and Hoppe recommend that we increase s_{offset} by 1 and try again. We can also increase the size of the hash table (so there are more empty slots). Note: To ensure that entries that collide in the offset table do not collide in the hash table, s_{hash} and s_{offset} should not share any common factors. Here's a sample command line to construct a perfect hash function and perform compression:

```
a.exe compress input.ppm occupancy.pbm data.ppm offset.offset
```

Viewing `.ppm`, `.pbm`, and `.offset` Images

The input, hash data, and output files are stored in the standard "Portable Pixel Map" format, `.ppm`. The occupancy file is stored as a "Portable Bit Map", `.pbm`. Many standard image viewing programs (Photoshop, GIMP, `xv`) will load and display these images. On UNIX/Cygwin, ImageMagick can be used to convert between image formats, such as the popular "Portable Network Graphics" format, `.png`. For example:

```
convert.exe tmp.ppm tmp.png
```

The offset image is stored in a *non-standard*, custom binary format, with a filename ending in `.offset`. Visualizing the offset image is not necessary to complete the assignment. But it might be helpful while debugging to view these images. Thus, we provide a simple OpenGL image viewer that will load `.ppm`, `.pbm`, and `.offset` images. If the machine you're working on does not have OpenGL and GLUT, you will need to download these libraries from http://www.opengl.org. We have provided a Makefile to build the ImageViewer executable. On Linux or FreeBSD, type: "`make unix`". On Mac OSX, type: "`make osx`". On Windows/Cygwin, if you have the files `OpenGL32.lib` and `glut32.lib`, type: "`make cygwin_lib`"; if you have `libopengl32.a` and `libglut32.a`, type: "`make cygwin_a`"; and if you have `libopengl.a` and `libglut.a`, type: "`make cygwin_x`". You may need to adjust the `CC`, `INCLUDE_PATH`, and `LIB_PATH` lines of the Makefile for your particular installation. Then to run the program type:

```
ImageViewer.exe lightbulb.ppm
```

Once the ImageViewer program is running, you can re-load the image by pressing '`r`'. Pressing '`q`' will quit the program.

Submission

Do all of your work in a new folder named `hw9` inside of your CSII homeworks directory. Please use the provided template `README.txt` file for any notes you want the grader to read. **You must do this assignment on your own, as described in the "Academic Integrity for Homework" handout. If you did discuss the problem or error messages, etc. with anyone, please list their names in your `README.txt` file.** When you are finished please zip up your folder exactly as instructed for the previous assignments and submit it through the course webpage. *Important Note:* Do not include the `image_viewer.cpp` file with your submission. The server compiles all `.cpp` files and including this file will cause an error since it contains its own `main()` function.