CSCI-1200 Data Structures — Fall 2022
Homework 9 — Perfect Hashing

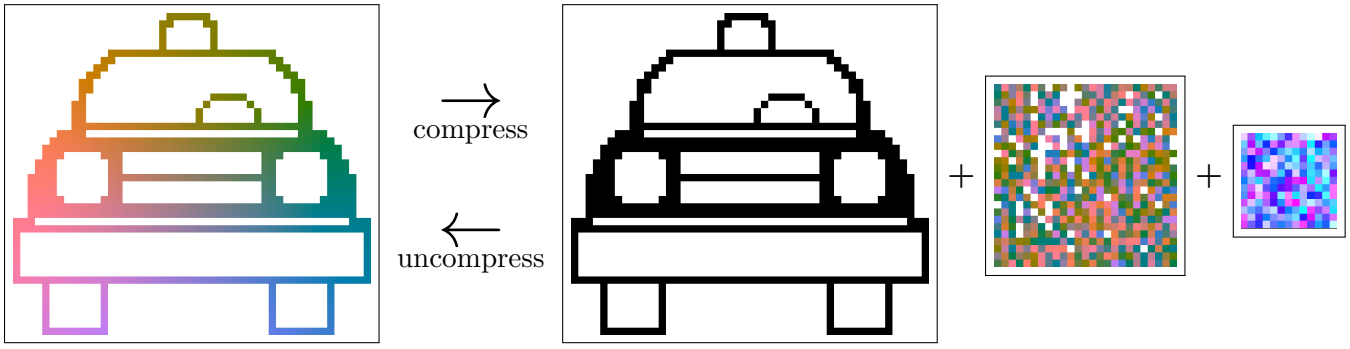
For this assignment you will implement a form of lossless image compression based on perfect hashing. A *perfect hash function* for a given set of keys maps those keys to the hash table with *no collisions*. This assignment is based on a paper from SIGGRAPH, a prestigious academic computer graphics conference:

"Perfect Spatial Hashing". S. Lefebvre, H. Hoppe. SIGGRAPH 2006, pages 579-588.
https://hhoppe.com/perfecthash.pdf
http://research.microsoft.com/en-us/um/people/hoppe/proj/perfecthash/

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 homework description 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 this 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 63% 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 an 8-bit number that 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:

```
hw9.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:

```
hw9.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.

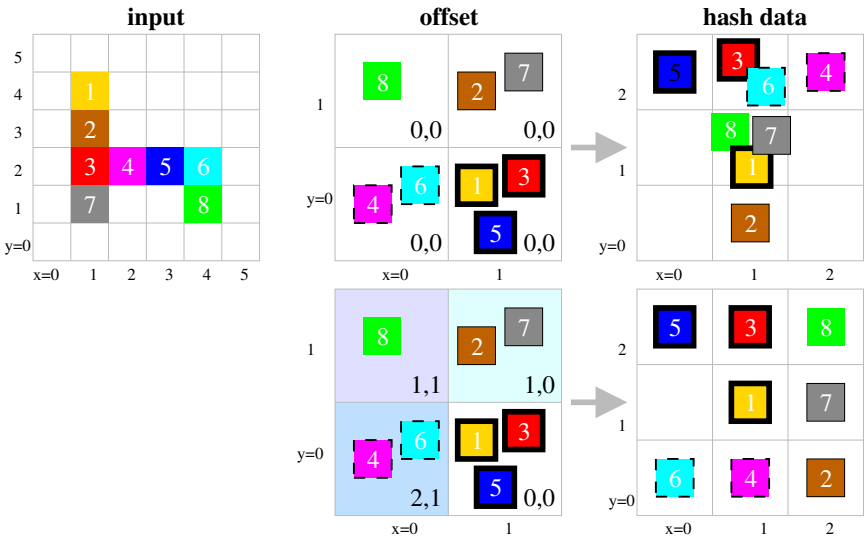
Constructing the Hash Function

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. However, the user *cannot* add new data pixels after the hash function is constructed, because we cannot guarantee a collision-free hashing of these new pixels.

Rather than representing the hash function as a mathematical formula or a simple C++ function, the hash function is stored as a lookup table of data, stored as an "offset image". What's a good size for the offset image? What's a good size for the hash data image? Find a perfect has is easier if the images are bigger, but to maximize compression, both images should be as small as possible. Furthermore, the hash data must have at least as many pixels as non-black pixels in the input. 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. Applying the hash lookup described above to the non-white pixels in the input image (shown in the top row below), we will get collisions in two of the hash data cells. Because we can only store one color per bin/pixel in the hash data image, this will not be a lossless representation of the input image.



Example from: http://research.microsoft.com/en-us/um/people/hoppe/perfecthash.pptx

So how do we assign offset values to separate these collisions? *NOTE: There are many valid answers for the offset image that create a perfect hash for this data... we just need to find one!* 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. So 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. *Note: There may be more than one choice!* And we continue to the next most populous offset table cell. 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 more expensive 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 (and repeat as necessary). We can also or alternatively increase the size of the hash table (so there are more empty slots). **IMPORTANT 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:

```
hw9.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*, 8-bit per pixel, custom binary format, with a filename ending in `.offset`. This format won't be recognized by any of image programs listed above. To visualize this data in the cyan/purple/blue images shown in this document, you will need to first convert the .offset file to a (larger) .ppm file with this command:

```
hw9.exe visualize_offset myfile.offset myfile.ppm
```

Provided Code and Homework Submission

Please study the provided code for parsing command line arguments and loading and saving image files with the templated `Image` class. Your task for this assignment is to complete the `Compress` and `UnCompress` functions. Be sure to write and use helper functions in your solution to keep your code organized and easy to understand and debug.

Use the provided template `README.txt` file to summarize the results of your implementation and your testing on a variety of image images. What compression ratio were you able to achieve with your solution? **You must do this assignment on your own, as described in the "Academic Integrity for Homework" handout. If you did discuss the assignment or debugging with anyone, please list their names in your `README.txt` file.**