CMSC 16200: Honors Introduction to Computer Science II

Homework 4 - Images Due: Wed, Mar 9th, 2022 @ 11:59 p.m.

1 Introduction

In this assignment, you will learn to implement Union Find algorithm for image processing. Throughout this homework, work **individually**, unless otherwise stated.

This assignment has a total of 92 points. **5 points are assigned based on program design and code style.**

2 Setup and Submission

As before, you can download the assignment tarball from Canvas and extract the files by typing

```
tar -xvf hw4-handout.tar.gz
```

- Some of the files worth looking at are listed below.
  Files you **should not** modify:

  ```
  img/*
  lib/*.h
  lib/*.c
  libpng_check.c
  Makefile
  ```

  Files you may modify (and the files denoted by ⋆ will be submitted for grading):

  ```
  ⋆ pixel.c
  ⋆ pixel.h
  pixel_test.c
  ⋆ graph.h (in case you want to define new graph data)
  ⋆ graph.c
  graph_test.c
  ⋆ unionfind.c
  ⋆ unionfind.h
  unionfind_test.c
  ⋆ segment.c
  ⋆ segment.h
  segment_test.c
  ⋆ theory.md
  ```

In order to submit your work (which as a reminder should be tested on the Linux machines), transfer your completed work to the Linux machines (if necessary). Create your hand-in file `hw4-handin.tgz` by running

```
make clean && make package
```

then move your created `.tgz` file into your SVN directory for this class. You can submit your .tgz file via

```
> svn add hw4-handin.tgz
> svn commit -m "hw4 complete"
```

If you are having problems with submitting your assignment, please come to office hours or post on Ed. Please do not wait until the due date to try to submit.

3 Image Processing

In this assignment, you will learn to process PNG images. PNGs are a very common lossless image compression file format.

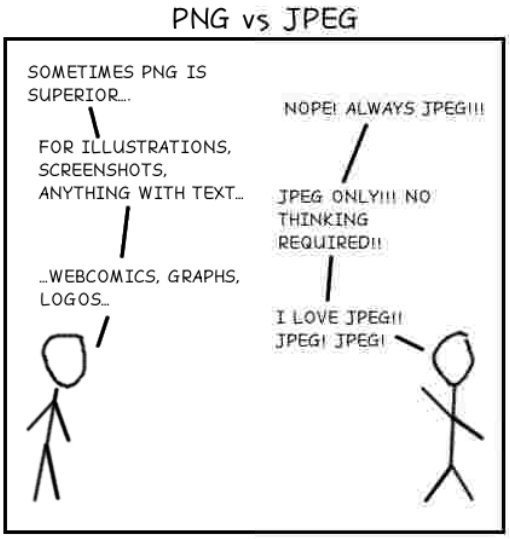


Figure 1: Source: http://lbrandy.com/blog/2008/10/my-first-and-last-webcomic/

3.1 Installation

You will be using the `libpng` [1] library to process PNG images using C. In order to compile your code, you will need a library called `libpng` installed on your system.

You can check if the library has been properly installed on your system by running the command we wrote for you:

```
>> make checklib
>> ./checklib
```

You may ignore the "unused parameter" warnings during this checking. If you have the library installed, the commands should print the following.

[1] http://www.libpng.org/pub/png/libpng.html

```
>> Reading from img/uchicago.png: Success!
>> Generating img/uchicago.html: Success!
```

Further, the command above should successfully generate the file `img/uchicago.html` that contains the uchicago shield logo. If the above commands were successful, then you may skip the remainder of this section.

If the library has **not** been installed, you have the following options:

- If you are running MacOS, which has Homebrew [2] installed, run:

  ```
  >> brew install libpng
  ```

- If you have root access on Linux or Bash for Windows, run:

  ```
  >> sudo apt install libpng-dev
  ```

- If you are running Cygwin, it might be a bit harder, but we suggest installing apt-cyg [3] and running:

  ```
  >> apt-cyg install libpng-devel
  ```

- If you are having trouble setting up the environment, it is suggested that you SSH into the CS department `linux.cs.uchicago.edu` machines, which already have this library installed.

3.2 Image_util

We have also provided you with a set of functions that read and write PNG files. Please check out `lib/image_util.h` and `lib/image_util.c`. You can freely invoke these functions in your code. The functions, however, are designed to only read certain types of PNG files. Below are some points worth noting

- The maximum size of the image is 1000 x 1000 pixels in terms of width and height. We define these limits using the constants `ROWS, COLS`. **The height refers to the number of rows, and the width refers to the number of columns**.
- The color space of the PNG file is *ARGB*. [4]

4 Strict Pixel Graph

In this assignment, we will apply the Union-Find technique to processing images represented by strict pixel graphs. One application is determining and labeling the connected components of a spatial data structure. Another application involves segmenting the pixels of an image into regions ("segmentation").

[2] http://brew.sh
[3] https://github.com/transcode-open/apt-cyg
[4] https://en.wikipedia.org/wiki/RGBA_color_model

4.1 Pixels

To capture the contents of a single pixel, we need two features: transparency and color. The *ARGB* representation stores both features.

We represent the transparency as an integer in the range [0, 256), where 0 is completely transparent, and 255 is completely opaque (non-transparent). This value is called represented by the alpha (A) component. The remaining three integers store the color, and each value is also in the range [0, 256). These values describe the intensity of red (R), green (G), and blue (B) in the pixel.

With this representation, we may describe a pixel using four integers between 0 (inclusive) and 256 (exclusive). One way to represent the four numbers is to hold them inside a 32-bit integer. We break the 32-bit value into four components with 8 bits each. Note that this packing is possible due to the limited range of pixel values.

$$a_0a_1a_2a_3a_4a_5a_6a_7r_0r_1r_2r_3r_4r_5r_6r_7g_0g_1g_2g_3g_4g_5g_6g_7b_0b_1b_2b_3b_4b_5b_6b_7$$

where:

- $a_0a_1a_2a_3a_4a_5a_6a_7$: represents the alpha value (how opaque the pixel is).
- $r_0r_1r_2r_3r_4r_5r_6r_7$: represents the intensity of the red component of the pixel
- $g_0g_1g_2g_3g_4g_5g_6g_7$: represents the intensity of the green component of the pixel
- $b_0b_1b_2b_3b_4b_5b_6b_7$: represents the intensity of the blue component of the pixel

As a reminder, each 8-bit component can range between a minimum of 0 (binary 00000000 or hex 0x00) to a maximum of 255 (binary 11111111 or hex 0xFF).

We have implemented a set of library functions that deals with the pixel representation above. It is recommended that you inspect the `get_red, get_green, get_blue, get_alpha,` and `make_pixel` functions provided in `pixel.c`.

4.2 Images

An image will be stored in a *two-dimensional* array of pixels, denoted by `pixels`. Thus, the pixel value of the point with `row` and `col` in the image will be stored at `pixels[row][col]`. Consider the simple digital image below, in which each pixel's color is represented by just a single letter for simplicity. Here we can imagine N = brown, G = gray, W = white, B = blue. Therefore,

```
pixels = {{B,N,N,G}, {B,W,W,W}, {G,G,W,G}, {N,N,N,G}}
```

For convenience, we will also keep an *one-dimensional* array `pixels` of pixels to represent the image. Pixels are stored in the array row by row, left to right starting at the bottom left of the image. For example, for the same image below, we have

```
pixels = {B,N,N,G,B,W,W,W,G,G,W,G,N,N,N,G}
```

For this task, you can compile and run your code using the commands below. We have provided some non-comprehensive tests in `pixel_test.c`.

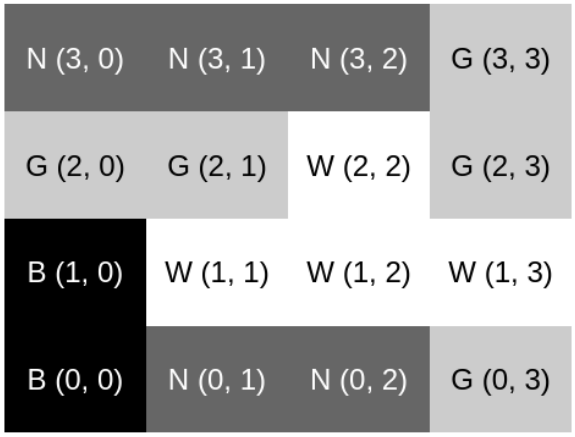


Figure 2: Sample image of 4×4 pixels. Coordinates are in (r, c) format.

```
> > make pixel
> > ./pixel
```

Task 4.1 (1 pts). In `pixel.c`, implement the function:

```
pixelID get_pixel_id(unsigned int row, unsigned int col, unsigned int width);
```

that returns its index to the 1-D pixel array, given the coords of the input pixel, `row` and `col`, and the width of the image `width`. In other words, this function should yield the following.

$$\texttt{pixels2d[row][col]} == \texttt{pixels1d[get\_pixel\_id(row, col, width)]}$$

Task 4.2 (1 pts). In `pixel.c`, implement the function:

```
unsigned int get_row(pixelID idx, unsigned int width);
```

that returns the row of the pixel, given its index `idx` and the width of the image `width`.

Task 4.3 (1 pts). In `pixel.c`, implement the function:

```
unsigned int get_col(pixelID idx, unsigned int width);
```

that returns the column the pixel, given its index `idx` and the width of the image `width`.

4.3 Graphs

Now we define the *strict pixel graph* of an image to be the pair $G = (V, E)$, where V is the set of pixels like those shown 2. The symbol E denotes the set of edges, where an edge e connects $v_0 = (x_0, y_0)$ with $v_1 = (x_1, y_1)$ provided $|x_0 - x_1| + |y_0 - y_1| = 1$ (i.e. v_0 and v_0 are adjacent and non-diagonal), and the colors of the two pixels are the same. When comparing colors, you *should* include the transparency values. The pixel graph is *undirected*.
For this section, you can compile and run your code using the following commands.

```
> > make graph
> > ./graph
```

Task 4.4 (3 pts). In `graph.c`, implement the function:

```
graph pixel_graph_new(unsigned int img_width,
                      unsigned int img_height,
                      pixel pixels[ROWS][COLS]);
```

that allocates enough space for the graph and initializes its required fields (see below for more details). You may assume that `img_width` and `img_height` do not exceed the maximum dimensions.

Task 4.5 (2 pts). In `graph.c`, implement the function:

```
void pixel_graph_free(graph G);
```

that frees the memory used by the graph.

Before you implement the above functions, you should first think about what data you want to store for the graph $G = (V, E)$. Fill in the corresponding fields in the file `graph.h`. You are free to design any data structure for representing the graph. The functions below may give you an idea about the types of operations your structure must support.

Task 4.6 (2 pts). In `graph.c`, implement the function:

```
bool is_vertex(graph G, pixelID idx);
```

that returns whether the given pixel ID corresponds to a valid vertex in the graph.

Task 4.7 (2 pts). In `graph.c`, implement the function:

```
bool are_neighbors(graph G, pixelID v, pixelID w);
```

that returns whether there is an edge between the given vertices `v` and `w`.

5 Union-Find

5.1 Finding Connected Components

In this section, you will implement a Union-Find data structure on a pixel graph. We can represent the Union-Find structure using an array `parentID[ROWS][COLS]`. For a given `row` and `col` (which correspond to a pixel ID), the value of `parentID[row][col]` will be the parent ID. If the current pixel is the root, then `parentID[row][col]` will be -1. Initially, before any of the Union operations, each element of the array should be -1, since every pixel is in its own subset.
Throughout this ask, you may assume that the given graph is non-null. You can compile and run your code using the following commands.

```
> > make unionfind
> > ./unionfind
```

Task 5.1 (5 pts). In `unionfind.c`, implement the function:

```
void init_union_find(graph G, pixelID parentID[ROWS][COLS]);
```

that initializes the Union-Find array `parentID` given the graph `G`. This function should create the structure such that each pixel is in its own subset. Remember that we have implemented functions for converting pixel indices from/to coordinates in the previous section. You should use these functions to follow the path from any pixel to its root repeatedly getting the parent node's `pixelID` from the `parentID` array.

Task 5.2 (10 pts). In `unionfind.c`, implement the function:

```
pixelID find(int parentID[ROWS][COLS], unsigned int width, pixelID idx);
```

that finds and returns the pixelID of the root of set with pixel as given by `pixelID idx`.

Task 5.3 (10 pts). In `unionfind.c`, implement the function:

```
void merge(int parentID[ROWS][COLS], unsigned int width, pixelID p1, pixelID p2);
```

that merges the two sets to which pixel `p1` and pixel `p2` belong, and it should make the one having the *smaller* `pixelID` value be the parent of the other. In case they already belong to the same group, make no changes and exit the function. You are **not** required to implement path compression.

You will now apply your Union-Find implementation to the problem of image analysis. Your code will merge groups of pixels so that each connected component of the strict pixel graph will represented by one set in Union-Find. To perform this task, you should scan the image considering all the pixel pairs for which edges exist (according to the pixel graph G). You should merge the sets of such pairs. Starting at $(r, c) = (0, 0)$, check to see if G contains an edge to $(r + 1, c) = (1, 0)$ and/or to $(r, c + 1) = (0, 1)$. If the edge exists, perform a union (`merge`) the two subsets. After processing this pixel, continue to the next row until all rows of pixels have been processed.

Task 5.4 (10 pts). In `unionfind.c`, implement the function:

```
void build(graph G, int parentID[ROWS][COLS]);
```

that does the operations described above.

5.2 Counting Connected Components

Your next task is to use your union-find data structure to find the number of connected components in the pixel graph. For the following two functions, you may assume that the given graph is non-null. Furthermore, you should assume that the `parentID` argument is the result after calling the `build` function (task 5.4). Thus, it "correctly" represents the union-find structure for the given graph.

Task 5.5 (5 pts). In `segment.c`, implement the function:

```
int count_connected_components(graph G, int parentID[ROWS][COLS]);
```

that counts the number of connected components in the graph.

5.3 Recoloring Images

In this task, you will use recolor an image using its connected components. To perform this operation, you can use the following idea for each pixel.

Write code for another scan of the image, this time doing the following for each pixel:

- Finding the root of the pixel's connected component
- Looking up the component number for that root (in a constructed hash table or list). We will denote the resulting component number by k.
- Determine the k^{th} color by calling the provided function `get_color(k)` in `lib/colors.h`.
- Reassign the pixel with the new color.

Task 5.6 (5 pts). In `segment.c`, implement the function:

```
void recolor_connected_components(graph G, pixelID parentID[ROWS][COLS]);
```

that does the operations described above. The test file is configured to save the recolored image to the file `img/recolored.png`. You should visually check that your result shows color changes for each connected component.

6 True/False

For the following True/False questions, clearly state your choice and *explain your answer a few sentences*. Place your responses in the file `theory.md`.

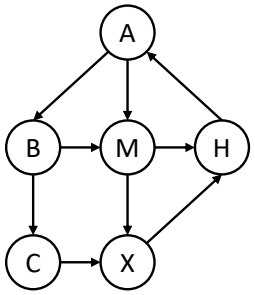
Task 6.1 (2 pts). TRUE or FALSE: Kruskal's algorithm for minimum spanning tree works with negative edge weights.

Task 6.2 (2 pts). TRUE or FALSE: One can find a maximum weight spanning tree of a connected graph by negating all of the edge weights and using a minimum spanning tree algorithm.

Task 6.3 (2 pts). TRUE or FALSE: The heaviest edge in a undirected weighted connected graph cannot belong to a minimum spanning tree.

7 DFS

A *DFS order* is a sequence of vertices of a graph in the order in which they are first visited by depth-first search (DFS). Consider the following graph:



One DFS order of this graph is $\{A, B, C, X, H, M\}$.

Task 7.1 (10 pts). Given the source vertex A, list ALL other possible DFS orders for this graph. Write your answer in the file `theory.md`.

8 Heapsort

We can use the invariant behind heap to implement an in-place sorting algorithm called heapsort. For simplicity, we use *max*-heaps, which satisfy:

- *Max-Heap Ordering Invariant*: Each node except for the root must be less or equal to its parent.

This guarantees that a *maximal* element is at the root of the heap, rather than a minimal one as we did in lecture.

The algorithm proceeds in two phases. In phase one we build up a heap spanning the whole array, in phase two we successively delete the maximum element from the heap and move it the end. Here is our implementation, written compactly. Note that we only sort the range $A[1, n)$, ignoring $A[0]$.

```
1  void heapsort(int[] A, int n)
2  //precondition: 1 <= n && n <= length(A);
3  //postcondition: is_sorted(A, 1, n);
4  {
5    int i;
6    for (i = 2; i < n; i++) {
7      percolate_up(A, i, i+1);
8    }
9    for (i = n-1; 2 <= i; i--) {
10     swap(A, 1, i);
11     percolate_down(A, 1, i);
12   }
13 }
```

The functions `percolate_up` and `percolate_down` are similar the functions presented in lecture. These functions take the heap array as the first argument, the current index of the node being percolated as the second argument, and the index right after the last element in the heap as the third argument. As a note, the function `is_sorted(A, lower, upper)` (referenced above) means that the range $A[lower, upper)$ is sorted in increasing order.

Task 8.1 (6 pts). Analyze the asymptotic complexity of our version of heapsort. You must provide a short justification. (Hint: Explain the number of times the functions `percolate_up` and `percolate_down` are called, as well as the running time of each function call.) Write your answer in `theory.md`.

9 Dijkstra

Task 9.1 (8 pts). Run Dijkstra's algorithm on the following graph.

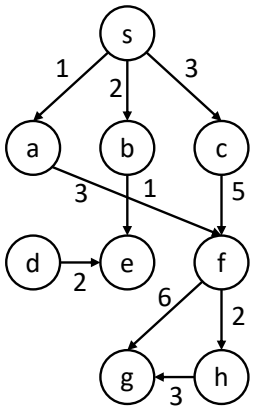


Figure 3: A weighted directed graph

Suppose we start Dijkstra's algorithm with the source vertex s. We can use this algorithm to find the shortest distance between the source and all reachable vertices. Fill in the table (in `theory.md`) that lists both the priority queue and the visited vertices at each step of Dijkstra's algorithm. You should break ties alphabetically based on the node's label (lower first). Write each vertex in the format of a (key, value) pair, where the key is the distance from s, and the value is the vertex label. In the file `theory.md`, we have completed the first few cells for you. You may not need all of the available slots.