Assignment 5 – Color Blindness Simulator

Mateo Garcia

CSE 13S – Spring 2023

Purpose

The purpose of this program is to be able to compress a file using Huffman Code. A file's contents are made up of bits, little 1's and 0's that make a pattern and that pattern makes a certain symbol. The series of 1's and 0's is called binary, and files are made up of binary at the lowest level. The amount of space a file takes up in a computer is usually the the size of the file (how many 1's and 0's there are), and a couple of other things. In order to make the file take up less space on the computer, we can use Huffman Coding. The Huffman Code takes the symbols that show up most frequently in the file, and switches their representation to use fewer than 8 bits, that means to use fewer than eight 1's and 0's for that symbol. To compensate and balance the changing of the file, the less common symbols will be switched to a representation that use more than 8 bits. After encoding the file using Huffman Coding, there will be fewer total 1's and 0's that are needed to represent the entire file, therefore compressing it.

How to Use the Program

This program uses a shell script. The shell script looks into the **files** directory which is where all the files to be encoded are, and runs the command **./huff** on the files whose name ends in **-.txt**. The corresponding output file after the script has been run has the same base but ends in **-.huff**. The script is called **runtests.sh** and if you go into the **cse13s/asgn6/** directory you can find it there.

First you want to run **make** and then **./runtests.sh**. That will run **./huff** on all of the test pictures. If you want to individually run **./huff**, you should run **./huff -** [***flag1***] [***flag2***]. There are three flags that can be added after the initial command.

The following are the types of flags you can add:

- -i : Sets the name of the input file. Requires a filename as an argument.
- -o : Sets the name of the output file. Requires a filename as an argument.
- -h : Prints a help message to **stdout**.

Program Design

Data Structures

This assignment will use the Unix file-I/O functions to read and write out bits for the files and will build upon the functions we wrote for Assignment 5. That will be the "bit writer" ADT. As for the Huffman Tree, it will use a binary tree ADT that uses Priority Queue, and Nodes. The priority queue is another ADT.

Algorithms

There are a couple algorithms used int his assignment. The first one is writing the tree, and another is compress the file using the tree.

```
Creating tree algorithm

Create and fill a PQ

Run the HC algorithm
    while Priority Queue has more than one entry
        Dequeue into left
        Dequeue into right
        Create a new node with a weight = left->weight + right->weight
        node->left = left
        node->right = right
        Enqueue the new node

Dequeue the queue's only entry and return it
```

```
Huff Compress File

huff_compress_file(outbuf, inbuf, filesize, num_leaves, code_tree, code_table)
for every byte b from inbuf
    code = code_table[b].code
    code_length = code_table[b].code_length
    for i = 0 to code_length - 1
        /* write the rightmost bit of code */
        /* prepare to write the next bit */

huff_write_tree(outbuf, node)
    if node is an internal node
        huff_write_tree(node->left)
        huff_write_tree(node->right)
    else
        node is a leaf
```

Function Descriptions

Constructor for Priority Queue:

```
/* put in pq.h */
typedef struct PriorityQueue PriorityQueue;
/* put in pq.c */
typedef struct ListElement ListElement;
struct ListElement {
    Node *tree;
    ListElement *next;
};
struct PriorityQueue {
    ListElement *list;
};
```

Functions for Priority Queue:

- **PriorityQueue *pq_create(void)** Allocate a **PriorityQueue** object and return a pointer to it
- **void pq_free(PriorityQueue **q);** Call **free()** on *q, and then sets *q to NULL

- **bool pq_is_empty(PriorityQueue *q);** Returns true if the queue's **list** field is NULL.
- **bool pq_size_is_1(PriorityQueue *q);** Returns true if queue contains a single value
- **void enqueue(PriorityQueue *q, Node *tree);** Insert a tree into the prio. queue. Keeps the tree with the lowest weight at the head.
- **bool dequeue(PriorityQueue *q, Node **tree)** If the queue is empty, return false. Otherwise removes the queue element with the lowest weight, set e to point to it, set parameter *tree = e-¿tree, call free(e) and return true.
- **void pq_print(PriorityQueue *q)** Prints the trees of the queue q.

Results

Have not started yet

Error Handling

Figure 1: Screenshot of scanbuild results