Jack McVeigh

CS 260

14 November 2021

Programming Assignment 3: Huffman Encoding

Introduction

Huffman Encoding is a lossless data compression method that bases its encoding on the frequencies of the data occurring in the original sample. For example, when encoding text, characters that are more frequent than others are assigned codes consisting of less bits. Files encoded with Huffman encoding are decoded using a code table that must be kept with the encoded file. This project will implement Huffman encoding for text files by implementing Heaps (Priority Queues) and Binary Search Trees (BSTs).

Encoding

A. Counting Character Frequencies from File

To generate the collection of counter structs from a given file, a hash table is used to easily increment counters for each character. A character's hash is simply its ASCII value, this allows for all entries to be unique and for each entry to be related to a character. From the hash table, a counter collection is then taken by only storing the characters with a frequency in the counter hash table.

```c
counterCollection *counterCollection_from_file(char *file_name)
{
    counterHashTable *cht = counterTable_from_file(file_name);
    counterCollection *cc = counterCollection_from_counterHashTable(cht);

    destroy_counterTable(cht);
    return cc;
}

counterCollection *counterCollection_from_counterHashTable(counterHashTable *cht)
{
    int i, j;
    counterCollection *collection = init_counterCollection();

    collection->size = 0;
    for (i = 0; i < cht->size; i++)
        if (cht->table[i])
            collection->size++;

    collection->entries = malloc(collection->size * sizeof(*collection->entries));
    j = 0;
    for (i = 0; i < cht->size; i++) {
        if (cht->table[i]) {
            collection->entries[j].character = (char)i;
            collection->entries[j].frequency = cht->table[i];
            collection->entries[j].id = j;
            j++;
        }
    }

    return collection;
}
```

Figure 1. Creating the collection of counter structs.

B. Creating a “Forest” of Min-Heaps from Frequencies

The heap is generated by storing trees containing counter data. The trees are simply inserted into the heap via the min heap strategy. This process is shown in lines 21-32 below.

```
21  tree *huffman_tree_from_counterCollection(counterCollection *cc)
22  {
23      int i, entry_count;
24      tree *t1, *t2, *joint;
25      heap *root;
26
27      root = init_heap(cc->size);
28      for (i = 0; i < cc->size; i++) {
29          t1 = init_tree();
30          t1->data = cc->entries[i];
31          insert_into_heap(root, t1);
32      }
33
34      entry_count = root->size - 1;
35      for (i = 0; i < entry_count; i++) {
36          t1 = poll_heap(root);
37          t2 = poll_heap(root);
38
39          joint = init_tree();
40          joint->left = t1;
41          joint->right = t2;
42
43          joint->data.id = -1;
44          joint->data.frequency = t1->data.frequency + t2->data.frequency;
45          insert_into_heap(root, joint);
46      }
47
48      /* return the huffman tree. (located at root) */
49      return poll_heap(root);
50  }
```

Figure 2. Lines 21-34 populate the heap "forest."

C. Forming the Huffman Tree

The Huffman tree is generated by popping the top two nodes from the heap, summing their frequencies to create a new node, and making the popped nodes the left and right

children of the newly formed intermediate node. This process is repeated until the tree is the last node available in the heap.

```
21  tree *huffman_tree_from_counterCollection(counterCollection *cc)
22  {
23      int i, entry_count;
24      tree *t1, *t2, *joint;
25      heap *root;
26
27      root = init_heap(cc->size);
28      for (i = 0; i < cc->size; i++) {
29          t1 = init_tree();
30          t1->data = cc->entries[i];
31          insert_into_heap(root, t1);
32      }
33
34      entry_count = root->size - 1;
35      for (i = 0; i < entry_count; i++) {
36          t1 = poll_heap(root);
37          t2 = poll_heap(root);
38
39          joint = init_tree();
40          joint->left = t1;
41          joint->right = t2;
42
43          joint->data.id = -1;
44          joint->data.frequency = t1->data.frequency + t2->data.frequency;
45          insert_into_heap(root, joint);
46      }
47
48      /* return the huffman tree. (located at root) */
49      return poll_heap(root);
50  }
```

Figure 3. Lines 35-50 generate the Huffman Tree.

D. Compile the Huffman Codes for Each Character

The Huffman Codes are generated recursively by walking the tree and setting the code value for a value when reaching its leaf. The id value stored in the counter struct data is used to simplify the mapping into the codes array.

```c
void huffman_code_walk(tree *root, int gen, char *code, huff_code *codes)
{
    /* If current node is a leaf */
    if ((!root->left) && (!root->right)) {
        code[gen] = '\0';

        codes[root->data.id].data = root->data;
        codes[root->data.id].code = strdup(code);
    }

    /* Traverse down edges */
    if (root->left) {
        code[gen] = '0';
        huffman_code_walk(root->left, gen+1, code, codes);
    }

    if (root->right) {
        code[gen] = '1';
        huffman_code_walk(root->right, gen+1, code, codes);
    }
}

huff_code *huffman_compile_codes(tree *root, int size)
{
    char code[ASCII_CHARACTER_COUNT] = "";
    huff_code *codes = malloc(size * sizeof(*codes));
    huffman_code_walk(root, 0, code, codes);
    return codes;
}
```

Figure 4. Compilation of the Huffman Codes.

E. Writing the Encoded Text and Code Table Files

The encoded file is written by reading each character and retrieving its code from the codes array. When the code is retrieved, the code is written to the new, encoded file.

```
int huffman_write_encoded_file(tree *root, huff_code *codes, int codes_size,
                               char *text_file_name, char *en_file_name)
{
    int i, encoded_size;

    FILE *text_fp = fopen(text_file_name, "r");
    if (!text_fp) {
        perror("Failed to open file");
        exit(EXIT_FAILURE);
    }

    FILE *en_fp = fopen(en_file_name, "w");
    if (!en_fp) {
        perror("Failed to open file");
        exit(EXIT_FAILURE);
    }

    char c;
    encoded_size = 0;
    while((c = fgetc(text_fp)) != EOF) {
        for (i = 0; i < codes_size; i++) {
            if (codes[i].data.character == c) {
                fprintf(en_fp, "%s", codes[i].code);
                encoded_size += strlen(codes[i].code);
            }
        }
    }

    fclose(text_fp);
    fclose(en_fp);
    return encoded_size;
}
```

Figure 5. Writing the Encoded Text to a file.

The code table file is simply written by recursing through the code table and writing each code's data to the file.

```
void huffman_write_code_table_file(huff_code *codes, int codes_size, char *ct_file_name)
{
    int i;

    FILE *ct_fp = fopen(ct_file_name, "w");
    if (!ct_fp) {
        perror("Failed to open file");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < codes_size; i++) {
        if (codes[i].data.frequency) {
            fprintf(ct_fp, "%c;%s;%d\n", codes[i].data.character, codes[i].code, codes[i].data.frequency);
        }
    }

    fclose(ct_fp);
}
```

Figure 6. Writing the Code Table to a file.

Decoding

A. Reading the Code Table File

The code table file is read like any other csv file, except it is delimited using semi-colons. The code table is populated line by line until all codes have been read.

```
huff_code *read_codes_from_code_table_file(char *file_name, int *codes_size)
{
    *codes_size = get_file_line_count(file_name);
    huff_code *codes = malloc(*codes_size * sizeof(*codes));

    FILE *ct_fp = fopen(file_name, "r");
    if (!ct_fp) {
        perror("Failed to open file");
        exit(EXIT_FAILURE);
    }

    int i = 0;
    char line[ASCII_CHARACTER_COUNT];
    while (fgets(line, ASCII_CHARACTER_COUNT, ct_fp))
    {
        char* tmp = strdup(line);
        codes[i].data.character = strtok(tmp, ";")[0];
        codes[i].code = strdup(strtok(NULL, ";"));
        codes[i].data.frequency = atoi(strtok(NULL, ";"));
        free(tmp);
        i++;
    }

    return codes;
}
```

Figure 7. Reading the Code Table from a file.

B. Decoding the Text File

Due to the variable length of codes, the encoded file must be read bit by bit into a temporary buffer until a code is matched. Once a code is matched, the buffer is cleared, and the next bit is read in the same process. This process continues until the entire file is read.

```c
void huffman_write_decoded_file(huff_code *codes, int codes_size,
                                char *en_file_name, char *text_file_name)
{
    FILE *en_fp = fopen(en_file_name, "r");
    if (!en_fp) {
        perror("Failed to open file");
        exit(EXIT_FAILURE);
    }

    FILE *text_fp = fopen(text_file_name, "w");
    if (!text_fp) {
        perror("Failed to open file");
        exit(EXIT_FAILURE);
    }

    char c;
    int i, found, gen = 0;
    char code[ASCII_CHARACTER_COUNT];
    while((c = fgetc(en_fp)) != EOF) {
        found = 0;
        code[gen] = c;
        code[gen+1] = '\0';
        for (i = 0; i < codes_size; i++) {
            if (!strcmp(code, codes[i].code)) {
                fprintf(text_fp, "%c", codes[i].data.character);
                gen = 0;
                found = 1;
                break;
            }
        }

        if (!found)
            gen++;
    }

    fclose(en_fp);
    fclose(text_fp);
}
```

Figure 8. Writing the decoded text to a file.

Conclusion

Huffman Encoding is a useful algorithm for encoding text and can be combined with other methods to achieve an even higher level of compression. The implementation described in this report achieves the ideal compression rate of the algorithm, while having decent running time.

The Huffman Encoding algorithm used makes efficient use of complex data types such as Heaps (Priority Queues) and Binary Search Trees to achieve quick running time. The frequency counter step of the encoder achieves O(n), the Huffman Tree generation achieves O(n), and the Huffman Code compilation achieves O(n), resulting in the final runtime of the Huffman encoding implementation being O(n).

This project was an effective problem for supporting the lecture material containing Heaps and BSTs.