Assignment 3
Algorithm Design and Analysis

bitjoy.net

November 6, 2015

I choose problem 2,3,5.

2 Greedy Algorithm

2.1 Algorithm description

We sort the jobs in descending finishing time f_i, and run them one by one on supercomputer. In this schedule, we will get a minimum completion time.

Let array J be the jobs array, J_i needs p_i seconds of time on the supercomputer, followed by f_i seconds of time on a PC. The pseudo-code like this:

```
JOBS-SCHEDULING(J)
1  sort J in descending finishing time f_i
2  run jobs in J one by one on supercomputer
```

2.2 Correctness of the algorithm

We can prove that algorithm JOBS-SCHEDULING can find an optimal schedule G by *exchange argument*.

For any given schedule $H \neq G$, we can repeatedly exchange **adjacent** jobs so as to convert H to G without increasing the completion time.

Suppose there are two adjacent jobs J_i and J_j in schedule H, $i < j$ and $f_i < f_j$. Before exchanging, J_i is first performed on supercomputer($S.comp$), followed by J_j. As soon as J_i is finished on $S.comp$, it is shifted onto PC_i, so is J_j. The process shows on Figure 1(a).

Let H' be the new schedule after exchanging J_i and J_j, as J_i and J_j is adjacent, all jobs except J_i and J_j are finished on the same time as in schedule H. The time J_i finished on $S.comp$ in H' is the same as the time J_j finished on $S.comp$ in H, say $t'_1 = t_1$ in Figure 1. But $f_i < f_j$, so J_i will be finished earlier in S' than J_j would be finished in S, say $t'_2 < t_2$.

So, our exchanged schedule H' doesn't have a greater completion time than H, that's say schedule G is the optimal schedule.

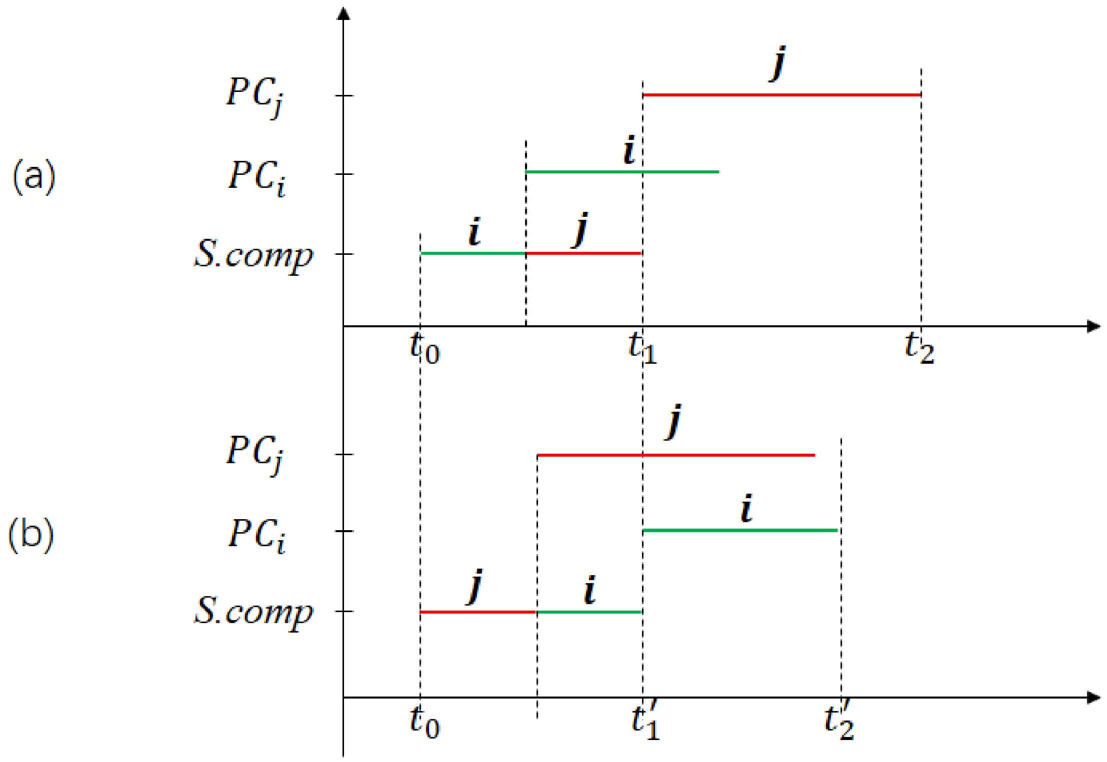

Figure 1: Jobs process before (a) and after (b) exchanging J_i and J_j.

2.3 Complexity of the algorithm

Suppose there are n jobs in J. In algorithm JOBS-SCHEDULING, we first sort all jobs, then run it one by one on supercomputer, so the time complexity is $O(nlgn)$.

3 Greedy Algorithm

3.1 Algorithm description

Let array B and G be the height of boys and girls respectively. We first sort B and G in descending order, then combine b_i and g_i as a pair. In this way, we will get minimum $\frac{1}{n}\sum_{i=1}^{n}|b_i - g_i|$. The pseudo-code like this:

MIN-HEIGHT-DIFF(B, G)

sort B in descending order
sort G in descending order
for $i = 1$ **to** n
    combine b_i and g_i as a pair

3.2 Correctness of the algorithm

We can prove that algorithm MIN-HEIGHT-DIFF can find an optimal matching solution S by *exchange argument*.

For any given solution $S' \neq S$, we can repeatedly exchange two pairs so as to convert S' to S without increasing the average difference.

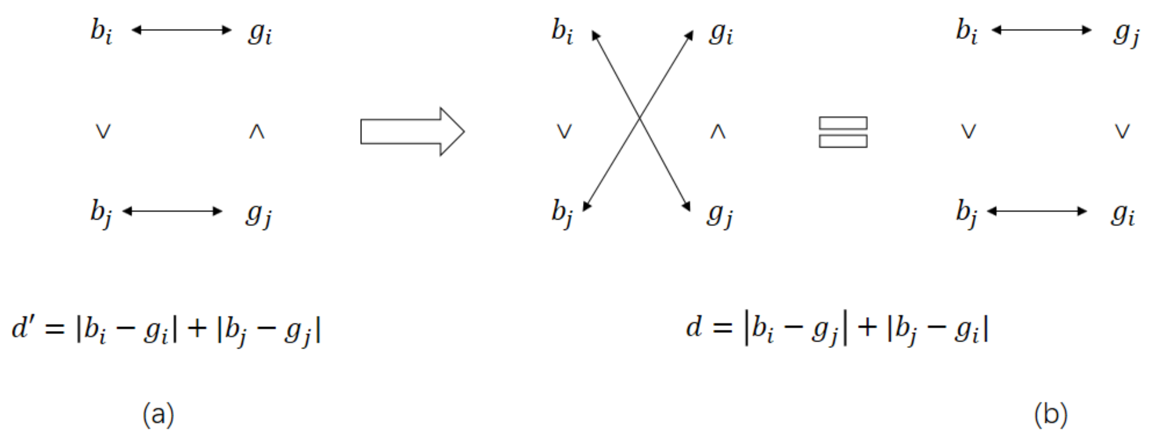

Figure 2: Matching before (a) and after (b) exchange pair $< b_i, g_i >$ and $< b_j, g_j >$, we have $d \leq d'$.

Before exchanging, we have two pairs $< b_i, g_i >$ and $< b_j, g_j >$ which satisfy $b_i > b_j$ and $g_i < g_j$ ($b_i < b_j$ and $g_i > g_j$ should be the same). The total difference of them is $d' = |b_i - g_i| + |b_j - g_j|$.

After exchanging, we have two new pairs $< b_i, g_j >$ and $< b_j, g_i >$. The new difference is $d = |b_i - g_j| + |b_j - g_i|$. So we have to prove that $d \leq d'$.

As $b_j < b_i$ and $g_i < g_j$, there are total six order sequence of them:

1. $g_i < g_j < b_j < b_i$
2. $g_i < b_j < g_j < b_i$
3. $g_i < b_j < b_i < g_j$
4. $b_j < g_i < g_j < b_i$
5. $b_j < g_i < b_i < g_j$
6. $b_j < b_i < g_i < g_j$

As for case 1, $d' = b_i - g_i + b_j - g_j$ and $d = b_i - g_j + b_j - g_i$, so $d = d'$. Similarly, $d = d'$ for case 6 and $d < d'$ for case 2,3,4,5. So, we have $d \leq d'$. That's to say, for any given solution $S' \neq S$, we can convert it to S without increasing the average difference, so solution S is the optimal solution.

3.3 Complexity of the algorithm

Suppose there are n boys and n girls, according to the pseudo-code, we have to sort twice and scan once, so the time complexity is $O(nlgn)$.

5 Programming

I implemented the *Huffman code* compression algorithm in C++, huffman code information is contained in the compressed file and the compression is lossless.

Here is the code, followed by detailed results.

```
#include<iostream>
#include<string>
#include<fstream>
#include<vector>
#include<map>
#include<list>
using namespace std;
const int MAX_LEN = 10*1024*1024; // read 10MB every time
const unsigned char MARKS[8] = { 0x80,0x40,0x20,0x10,0x8,0x4,0x2,0x1 }; // x&MARKS[i] to get ith bit
class HuffmanCode
{
public:
    HuffmanCode() {};

    //count letter frequency in file src
    void CountLetter(string src)
    {
        ifstream is(src, ios::binary);
        char *buf = new char[MAX_LEN];
        while (is.peek() != EOF)
        {
            is.read(buf, MAX_LEN);
            int len = is.gcount();
            for (int i = 0; i < len; i++)
                letter_count[buf[i]]++;
        }
        is.close();
        delete[] buf;

        map<char, int>::iterator it = letter_count.begin();
        while (it != letter_count.end())
        {
            Node nd(-1, true, it->first, -1, -1, -1);
            count_node.insert(pair<int, Node>(it->second, nd));
            it++;
        }
    }

    void ConstructHuffmanTree()
    {
        huffman_tree.resize(letter_count.size() * 2 - 1); // n=2n_0-1
        int k = 0;
        multimap<int, Node>::iterator it1, it2;
        while (count_node.size() > 1)
        {
            it2 = count_node.begin();
            it1 = it2;
            it2++;
            if ((it1->second).is_leaf)
            {
                (it1->second).id = k;
                (it1->second).parent = k + 1;
                huffman_tree[k++] = it1->second;
            }
            else
                huffman_tree[(it1->second).id].parent = k;

            int p = huffman_tree[(it1->second).id].parent;
```

```
            if ((it2->second).is_leaf)
            {
                (it2->second).id = p + 1;
                (it2->second).parent = p;
                huffman_tree[p + 1] = it2->second;
                k = p + 2;
            }
            else
            {
                huffman_tree[(it2->second).id].parent = p;
                k = p + 1;
            }
            Node pnd(p, false, ' ', -1, (it1->second).id, (it2->second).id);
            huffman_tree[p] = pnd;
            count_node.insert(pair<int, Node>(it1->first + it2->first, pnd));
            count_node.erase(it1);
            count_node.erase(it2);
        }
        it1 = count_node.begin();
        huffman_tree[(it1->second).id].parent = -1; // root of huffman tree
    }

    void GenerateHuffmanCode()
    {
        for (int i = 0; i < huffman_tree.size(); i++)
        {
            if (huffman_tree[i].is_leaf)
            {
                vector<char> inverse_code;
                int j = i, k;
                //get inverse huffman code by backtracing
                while (huffman_tree[j].parent != -1)
                {
                    k = huffman_tree[j].parent;
                    if (huffman_tree[k].lchild == j)
                        inverse_code.push_back('0'); // 0 for left
                    else
                        inverse_code.push_back('1'); // 1 for right
                    j = k;
                }
                reverse(inverse_code.begin(), inverse_code.end());
                letter_hcode[huffman_tree[i].letter] = inverse_code;
            }
        }
    }

    //we first write huffmancode as meta data of compressed file
    void WriteHuffmanCode(ofstream &os)
    {
        map<char, vector<char>>::iterator it = letter_hcode.begin();
        int cnt = letter_hcode.size();
        os.write((const char*)&cnt, sizeof(int)); // number of leaf nodes
        while (it != letter_hcode.end())
        {
            os.write(&(it->first), sizeof(char));
            cnt = (it->second).size();
```

```
            os.write((const char*)&cnt, sizeof(int));
            os.write(&((it->second)[0]), (it->second).size()*sizeof(char));
            it++;
        }
        char c = '\n';
        cnt = -1;
        os.write(&c, sizeof(char)); os.write((const char*)&cnt, sizeof(int)); // end of huaffman code
    }

    void Compressing(string src, string dest)
    {
        ifstream is(src, ios::binary);
        ofstream os(dest, ios::binary);
        WriteHuffmanCode(os);
        char *is_buf = new char[MAX_LEN], *os_buf = new char[MAX_LEN];
        list<char> tmp_hcode;
        int start_pos = 0, i, j, k, len, t;
        char c;
        list<char>::iterator it;
        while (is.peek() != EOF)
        {
            is.read(is_buf, MAX_LEN);
            len = is.gcount();
            for (i = 0; i < len; i++)
                tmp_hcode.insert(tmp_hcode.end(), letter_hcode[is_buf[i]].begin(), letter_hcode[is_buf[i]].end());
            k = tmp_hcode.size() / 8;
            t = 0; i = 0;
            it = tmp_hcode.begin();
            while (i < 8 * k)
            {
                c = 0x0;
                for (j = i; j <= i + 7; j++)
                {
                    c = (*it == '1') ? (c | (1 << (i + 7 - j))) : c;
                    it++;
                }
                os_buf[t++] = c;
                i += 8;
            }
            os.write(os_buf, t*sizeof(char));
            tmp_hcode.erase(tmp_hcode.begin(), it);
        }
        c = 0x0;
        i = 7;
        bool done = true;
        while (it != tmp_hcode.end())
        {
            done = false;
            c = (*it == '1') ? (c | (1 << i)) : c; // left bits
            i--;
            it++;
        }
        if (!done)os.write(&c, sizeof(char));
        c = 7 - i;
        os.write(&c, sizeof(char)); // mark for the last byte.
        is.close();
```

```
        os.close();
        delete[] is_buf;
        delete[] os_buf;

    }

    void Compress(string src, string dest)
    {
        CountLetter(src);
        ConstructHuffmanTree();
        GenerateHuffmanCode();
        Compressing(src, dest);
    }

    void InsertIntoHuffmanTree(char letter, string &code, int &k)
    {
        int parent = 0;
        for (int i = 0; i < code.size(); i++)
        {
            if (code[i] == '0'&&huffman_tree[parent].lchild == -1)
            {
                Node nd(k, false, ' ', parent, -1, -1);
                huffman_tree[k] = nd;
                huffman_tree[parent].lchild = k;
                parent = k++;
            }
            else if (code[i] == '1'&&huffman_tree[parent].rchild == -1)
            {
                Node nd(k, false, ' ', parent, -1, -1);
                huffman_tree[k] = nd;
                huffman_tree[parent].rchild = k;
                parent = k++;
            }
            else parent = (code[i] == '0') ? huffman_tree[parent].lchild : huffman_tree[parent].rchild;
        }
        huffman_tree[parent].is_leaf = true;
        huffman_tree[parent].letter = letter;
    }

    void ConstructHuffmanTreeFromFile(ifstream &is)
    {
        char letter; int len;
        is.read((char*)&len, sizeof(int)); // first read number of leaf nodes
        huffman_tree.resize(2 * len - 1); // n=2n_0-1
        Node root(0, false, ' ', -1, -1, -1);
        huffman_tree[0] = root;
        int k = 1;
        while(true)
        {
            is.read(&letter, sizeof(char));
            is.read((char*)&len, sizeof(int));
            if (letter == '\n'&&len == -1)break;
            string code(len,'\0'); // char *tmp = new char [len + 1]; tmp[len] = '\0';
            is.read(&code[0], len*sizeof(char));
            InsertIntoHuffmanTree(letter, code, k);
        }
```

```
    }
    void Decompressing(ifstream &is, ofstream &os)
    {
        char *is_buf = new char[MAX_LEN], *os_buf = new char[MAX_LEN];
        list<char> tmp_hcode;
        list<char>::iterator it1, it2;
        int len, i, j, p, t;
        bool last_read = false;
        char c;
        while (is.peek() != EOF)
        {
            is.read(is_buf, MAX_LEN);
            len = is.gcount();
            if (len < MAX_LEN)last_read = true;
            for (i = 0; i < len; i++)
            {
                if (last_read && (i == len - 2))break;
                c = (unsigned char)is_buf[i];
                for (j = 0; j < 8; j++)
                    tmp_hcode.insert(tmp_hcode.end(), '0' + ((c&MARKS[j]) >> (7 - j)));
            }
            if (last_read)
            {
                int b = is_buf[len - 1]; // only b bits in (len-2)th byte used
                c = is_buf[len - 2];
                for (j = 0; j < b; j++)
                    tmp_hcode.insert(tmp_hcode.end(), '0' + ((c&MARKS[j]) >> (7 - j)));
            }
            it1 = tmp_hcode.begin();
            t = 0;
            while (it1 != tmp_hcode.end())
            {
                p = 0;
                it2 = it1;
                while (!huffman_tree[p].is_leaf)
                {
                    p = (*it1 == '0') ? huffman_tree[p].lchild : huffman_tree[p].rchild;
                    it1++;
                    if (it1 == tmp_hcode.end())break;
                }
                if (huffman_tree[p].is_leaf)os_buf[t++] = huffman_tree[p].letter;
                if (it1 == tmp_hcode.end())
                {
                    if (huffman_tree[p].is_leaf)tmp_hcode.clear();
                    else tmp_hcode.erase(it2);
                    break;
                }
            }
            os.write(os_buf, t*sizeof(char));
        }
        delete[] is_buf;
        delete[] os_buf;

    }
```

```
    void Decompress(string src, string dest)
    {
        ifstream is(src, ios::binary);
        ofstream os(dest, ios::binary);
        ConstructHuffmanTreeFromFile(is);
        Decompressing(is, os);
        is.close();
        os.close();
    }

private:
    map<char, int> letter_count;
    typedef struct Node
    {
        int id;
        bool is_leaf;
        char letter;
        int parent, lchild, rchild;
        Node() {}
        Node(int i, bool il, char lt, int p, int lc, int rc)
            :id(i), is_leaf(il), letter(lt), parent(p), lchild(lc), rchild
    (rc) {}
    };
    multimap<int, Node> count_node;
    vector<Node> huffman_tree;
    map<char, vector<char>> letter_hcode; // hufman code for each letter
};
int main()
{
    //string src_file = "Aesop_Fables.txt";
    string src_file = "graph.txt";
    string compressed_file = "compressed.hzip";
    string decompressed_file = "decompressed.txt";
    HuffmanCode hc;
    hc.Compress(src_file, compressed_file);
    //hc.Decompress(compressed_file, decompressed_file);
    return 0;
}
```

The compression results are showed below:

Table 1: Compression results of my implementation

File	Size before compressed	Size after compressed	Compression ratio
Aesop_Fables.txt	186KB	107KB	57.53%
graph.txt	2046KB	910KB	44.48%

As the size of *Aesop_Fables.txt* is much smaller than *graph.txt*'s, so former compression ratio is bigger than latter's. What's more, 90% content of *graph.txt* is numbers, so the height of its huffman tree is smaller than *Aesop_Fables.txt*'s, more bytes will be converted to shorter bits, so the compression ratio is bigger.