

数据压缩实验报告

1 代码结构说明

本实验包含 Huffman 编码，算数编码，LZ 编码三种编码的编解码以及测试和评估的实现，使用的语言为 rust。

目录结构如下

```
/src
| /compresser
| | arithmetic.rs
| | huffman.rs
| | lz.rs
| | mod.rs
| data
| main.rs
```

`main.rs` 定义了主函数，用于测试各种编码的正确性压缩效率以及运行时间。

`data` 是用于压缩的测试文件，为纯文本类型。

`/compresser` 定义了压缩算法模块，其中 `mod.rs` 中定义了名为 `Compressor` 的 Trait，约束了所有压缩编码的调用接口。

```
pub trait Compressor {
    fn build(frequency: &Vec<(u8, u64)>) -> Self;
    fn encode(&self, s: &str) -> String;
    fn decode(&self, s: &str) -> String;
}
```

`arithmetic.rs`, `huffman.rs`, `lz.rs` 中的数据结构均实现了这个 Trait.

以上所有的代码具体实现可以见附件。

2 算法思路及实现

2.1 Huffman 编码

在 `Huffman.rs`，定义了以下数据结构

```
pub struct HuffmanCompressor {
    trie: [[usize; 2]; 256],
    dict: [String; 128]
}
```

其中 `trie` 是形如字典树的 Haffman 树，为了方便实现。定义第 255 号节点为根节点，0-127 号节点为叶节点代表各个字符，128-254 号为节点为内部节点池。`dict` 则是存放了每个字符到根节点路径上的字符串，即对应的编码。

为了构建 Haffman 树，首先得预处理数据，统计出各字符出现的频率。然后使用优先队列每次取出 Trie 树上频率最低的两个节点合并起来连接到新节点。最后树上 dfs 得到对应的每个字符对应的编码，插入到 `dict` 中。

```
fn build(frequence: &Vec<u8, u64>) -> Self {
    let mut trie = [[0; 2]; 256];
    let dict = [(); 128].map(|_| String::new() );

    let mut x = 127;

    let mut que : BinaryHeap<_> = frequence.iter()
        .map( |(p, q)| (- (*q as i64), *p) ).collect();

    while que.len() >= 2 {
        let u = que.pop().unwrap();
        let v = que.pop().unwrap();

        x += 1;

        trie[x][0] = u.1 as usize;
        trie[x][1] = v.1 as usize;

        que.push((u.0 + v.0, x as u8) );
    }

    trie[255] = trie[x];

    let mut ret = Self { trie, dict };

    ret.dfs(255, String::new());

    ret
}
```

dfs 的实现如下

```
fn dfs(&mut self, x: usize, s: String) {
    if x < 128 { self.dict[x] = s; }
    else { self.dfs(self.trie[x][0], s.clone() + "0" );
    self.dfs(self.trie[x][1], s + "1" ); }
}
```

编码时逐字符输出对应的编码

```
fn encode(&self, s: &str) -> String {
    s.as_bytes().iter().map(|&c| &self.dict[c as usize])
    .cloned().collect()
}
```

解码时，将 Huffman 树当成自动机，一旦识别到对应字符则输出，并且返回到根节点。

```
fn decode(&self, s: &str) -> String {
    let mut ans = String::new();
    let mut x = 255;
    for &u in s.as_bytes() {
        x = self.trie[x][u as usize - 48];
        x = if x <= 127 { ans.push(x as u8 as char); 255 } else {x}
    }
    ans
}
```

2.2 算数编码

由于算数编码在压缩的字符串较长时，会出现精度丢失问题，所以在这里引入了第三方的高精度整数库 `num_bigint`，同时将浮点运算转化为等效的整数运算。

在编码的过程中

- 首先假设原始区间为 $[l, r) \leftarrow [0, 1)$ 。
- 每输入一个字符，假设当前字符的前缀频数为 x_i ，后缀频数为 y_i ，则将区间扩张为 n 倍并且加上位移。
 $[l, r) \leftarrow [nl + (r - l) \times x_i, nr - (r - l) \times y_i)$ 。

最终得到了区间 $[l_n, r_n)$ ，设 $t = n^n$ ，二分答案找到最短的 01 编码 $c_1 \cdots c_k$ ，满足

$$l_n \leq \sum_{i=1}^k c_i \times \lfloor \frac{t}{2^i} \rfloor < r_n。$$

```
fn encode(&self, s: &str) -> String {
    let mut ans = String::new();
    let mut l = 0.to_biguint().unwrap();
    let mut r = 1.to_biguint().unwrap();
    let mut t = 1.to_biguint().unwrap();
    for c in s.as_bytes() {
        let (x, y) = self.segments.get(c).unwrap();
        let diff = &r - &l;
        l *= self.length.to_biguint().unwrap(); l +=
x.to_biguint().unwrap() * &diff;
        r *= self.length.to_biguint().unwrap(); r -=
(self.length.to_biguint().unwrap() - y.to_biguint().unwrap()) * &diff;
        t *= self.length.to_biguint().unwrap();
    }
    let mut sum = 0.to_biguint().unwrap();
```

```

    let mut bit = 1u64;
    while &sum < &l {
        let tmp = &sum + (&t >> bit);
        if tmp == 0.to_biguint().unwrap() {break;}
        if tmp < r { sum = tmp; ans += "1"; }
        else { ans += "0"; }
        bit += 1;
    }
    ans
}

```

解码时，首先假设当前区间为 $[l_0, r_n) = [0, n^n)$ ，计算出目标点为 $p = \sum_{i=1}^k c_i \times \lfloor \frac{t}{2^i} \rfloor$ ，循环 n 次，保持缩小区间时 $l_0 \leq l_1 \leq \dots \leq l_n \leq p < r_n \leq r_{n-1} \leq \dots \leq r_0$ ，每次输出满足条件的区间 $[l_i, r_i)$ 对应的字符即可。

```

fn decode(&self, s: &str) -> String {
    let mut ans = String::new();
    let mut t = self.length.to_biguint().unwrap().pow(self.n as u32);
    let mut res = 0.to_biguint().unwrap();

    let mut bit = 1u64;
    for &c in s.as_bytes() {
        if c == 49u8 { res += &t >> bit; }
        bit += 1;
    }
    let mut l = 0.to_biguint().unwrap();
    let mut w = 1.to_biguint().unwrap();
    for _ in 0..self.n {
        t /= self.length.to_biguint().unwrap();
        let p = (&res - &l) / (&w * &t);
        for (c, (x,y)) in self.segments.iter() {
            if x.to_biguint().unwrap() <= p && y.to_biguint().unwrap() > p
            {
                l = &l + &t * &w * x;
                w *= y - x;
                ans.push(*c as char);
                break;
            }
        }
    }
    ans
}

```

2.3 LZ 编码

LZ 编码是自适应编码，所以构建时只需要知道 $\lceil \log_2 m \rceil$ 即可， m 为最小分组长度。

编码时，首先建立一颗空的 Trie 树自动机（代码直接使用 BTreeMap 模拟了），每次识别到 Trie 之外的字符串，将该字符串分配新的编号并且加入 Trie 树，并且输出其父节点对应编号的二进制以及当前字符的二进制表

示。

```
fn encode(&self, s: &str) -> String {
    let mut ans = String::new();
    let mut map = BTreeMap::<String, usize>::new();

    map.insert(String::new(), 0);

    let mut now = String::new();
    let mut pre = 0usize;
    let mut cnt = 0;

    for &c in s.as_bytes() {
        now.push(c as char);
        if !map.contains_key(&now) {
            cnt += 1;
            ans += format!("{:0>11b}", pre).as_str();
            ans += format!("{:0>7b}", c).as_str();
            map.insert(now, cnt);
            now = String::new();
        }
        pre = *map.get(&now).unwrap();
    }

    ans
}
```

解码时，通过已知的分组长度，每次读取一组，一边解码一边还原 Trie 树。

```
fn decode(&self, s: &str) -> String {
    let mut ans = String::new();
    let block = 18;

    let mut vec = vec![String::from("")];

    for i in 0..s.len() / block {
        let x = i * block;
        let y = (i + 1) * block - 7;
        let z = (i + 1) * block;

        let pre = usize::from_str_radix(&s[x..y], 2).unwrap();
        let c = usize::from_str_radix(&s[y..z], 2).unwrap();

        let mut str = vec[pre].clone();
        str.push(c as u8 as char);

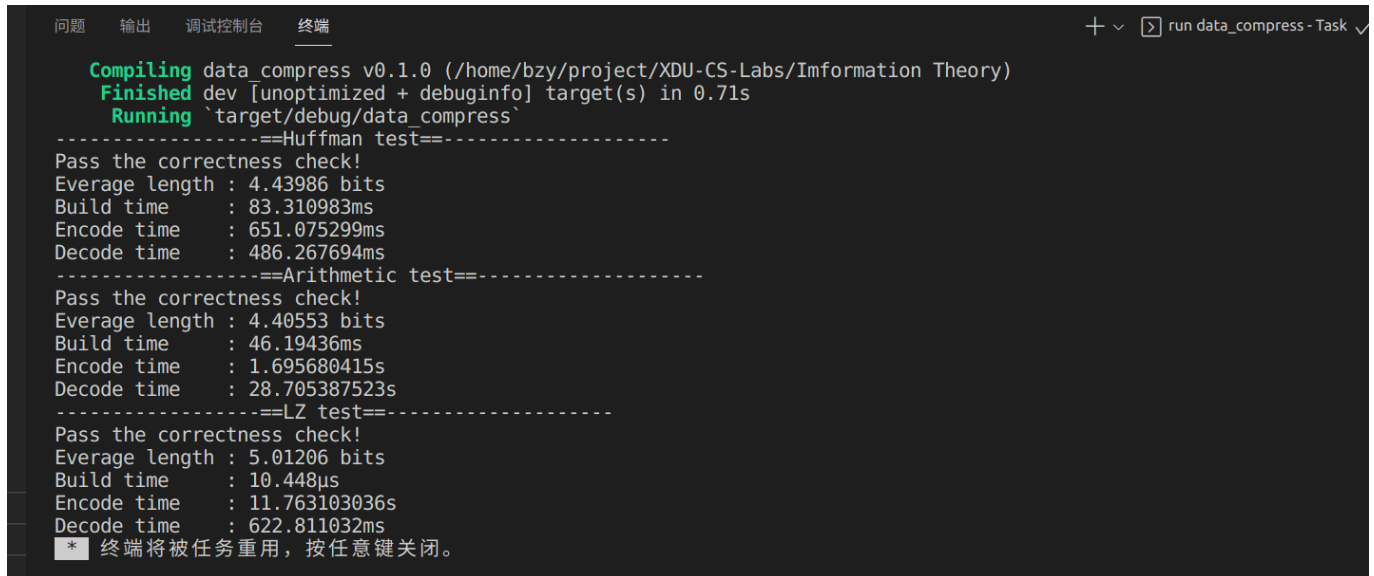
        ans += str.as_str();

        vec.push(str);
    }
}
```

```
    ans  
}
```

3 测试结果

在 `fn main()` 中，有着对三种编码实现的正确性检测以及 benchmark，结果如下



```
问题  输出  调试控制台  终端  + v [x] run data_compress - Task ✓  
  
Compiling data_compress v0.1.0 (/home/bzy/project/XDU-CS-Labs/Information Theory)  
Finished dev [unoptimized + debuginfo] target(s) in 0.71s  
Running `target/debug/data_compress`  
-----Huffman test-----  
Pass the correctness check!  
Everage length : 4.43986 bits  
Build time    : 83.310983ms  
Encode time   : 651.075299ms  
Decode time   : 486.267694ms  
-----Arithmetic test-----  
Pass the correctness check!  
Everage length : 4.40553 bits  
Build time    : 46.19436ms  
Encode time   : 1.695680415s  
Decode time   : 28.705387523s  
-----LZ test-----  
Pass the correctness check!  
Everage length : 5.01206 bits  
Build time    : 10.448µs  
Encode time   : 11.763103036s  
Decode time   : 622.811032ms  
[*] 终端将被任务重用，按任意键关闭。
```

其中 Huffman 编码和 LZ 编码均运行编解码 1000 次，算数编码仅运行 1 次。

可以看出算数编码的压缩效率最高，但是只比 Huffman 编码有微小提高，且运行时间远远大于其他两者；Huffman 编码编码效率以及运行效率都很高；LZ 编码编码效率和运行时间都略逊于 Huffman 编码，但是自适应编码的特性可以使得编码不需要额外的先验信息。