XCH 配平原理与算法

Ying Kanyang (LEXUGE) <LEXUGEyky@outlook.com>

February 20, 2018

Contents

1 XCH 简介

XCH 是自动解析并配平化学方程式的套件，由lib_xch与xch-ceb组成。使用 Rust 语言编写。其中所有的实现算法都在 lib_xch 中实现，并暴露 API，xch-ceb 是 lib_xch 的前端。目标是“轻巧，快速，安全”。

本文将介绍 XCH 的原理与算法，并相应地给出其实现。并给出目前存在的缺陷，以及未来的计划。

1.1 lib_xch

lib_xch 主要组成部分是 Delta-3 解析模块 (mod) 与 AlphaForce 配平模块。Delta-3 解析模块使用的 Regex 分词提取 Token 的方法，并将各元素在各分子式的个数转换为一张表 (table)。其中，使用了从内到外拆分的方式来实现多层括号的嵌套。作为解析模块的一部分，也作了相应的语法检查。

AlphaForce 配平模块先是将 Delta-3 解析模块的结果转换为方程组 (Equation Set)，并使用矩阵的方式表示出来，使用了高斯-约当消元算法 (The Gaussian-Jordan Algorithm) 解出各项系数，从而得出结果。

当然，由于化学方程式并不是都能转换为拥有唯一解的方程组，因此需要做一部分的优化以及处理。并且，由于算法本身的特殊性，计算过程中可能产生分数，需要设计专门的数据结构来处理，以达到“零精度损失”的目标。

1.2 xch-ceb

xch-ceb 是 XCH 的前端，负责调用 lib_xch 提供的 API，并处理用户的输入，最后返回 lib_xch 的结果。

2 原理及算法

2.1 原理

先来看一个未配平的化学方程式：

$$As_2O_3 + Zn + HCl = AsH_3 + ZnCl_2 + H_2O \tag{1}$$

设每个化学式 (Chemical Formula) 前的系数为未知数，可得：

$$x_1As_2O_3 + x_2Zn + x_3HCl = x_4AsH_3 + x_5ZnCl_2 + x_6H_2O \tag{2}$$

根据原子个数守恒 (the law of conservation of atoms)，可将上述方程转换为关于 x 的多元一次方程组：

$$\begin{cases} 2x_1 + 0x_2 + 0x_3 = 1x_4 + 0x_5 + 0x_6 \cdots As \\ 3x_1 + 0x_2 + 0x_3 = 0x_4 + 0x_5 + 1x_6 \cdots O \\ 0x_1 + 1x_2 + 0x_3 = 0x_4 + 1x_5 + 0x_6 \cdots Zn \\ 0x_1 + 0x_2 + 1x_3 = 3x_4 + 0x_5 + 2x_6 \cdots H \\ 0x_1 + 0x_2 + 1x_3 = 0x_4 + 2x_5 + 0x_6 \cdots Cl \end{cases} \tag{3}$$

整理得：

$$\begin{cases} 2x_1 - x_4 = 0 \cdots As \\ 3x_1 - x_6 = 0 \cdots O \\ x_2 - x_5 = 0 \cdots Zn \\ x_3 - 3x_4 - 2x_6 = 0 \cdots H \\ x_3 - 2x_5 = 0 \cdots Cl \end{cases} \tag{4}$$

显然，方程组存在多解。将其转换为增广矩阵的形式：

$$A = \left[\begin{array}{cccccc|c} 2 & 0 & 0 & -1 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & -3 & 0 & -2 & 0 \\ 0 & 0 & 1 & 0 & -2 & 0 & 0 \end{array}\right] \tag{5}$$

对其进行高斯-约当消元，得到：

$$A = \left[\begin{array}{cccccc|c} 1 & 0 & 0 & 0 & 0 & -\frac{1}{3} & 0 \\ 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & -3 & 0 & -2 & 0 \\ 0 & 0 & 0 & 1 & 0 & -\frac{2}{3} & 0 \\ 0 & 0 & 0 & 0 & 1 & -2 & 0 \end{array}\right] \tag{6}$$

自由元 (Free Variable)x_6，设 n 个自由元中任意一个为 1，其余为 0，并求出所有未知数。通过 n 组解的线性表示，就可以得出所有的可能解，也就是所有可能的系数。
这里，设所有的自由元均为 1[1]，得到解为：

$$\begin{cases} x_1 = \frac{1}{3} \\ x_2 = 2 \\ x_3 = 4 \\ x_4 = \frac{2}{3} \\ x_5 = 2 \\ x_6 = 1 \end{cases} \tag{7}$$

因为系数为整数，对其化整：

$$\begin{cases} x_1 = 1 \\ x_2 = 6 \\ x_3 = 12 \\ x_4 = 2 \\ x_5 = 6 \\ x_6 = 3 \end{cases} \tag{8}$$

所以，解为：

$$As_2O_3 + 6Zn + 12HCl = 2AsH_3 + 6ZnCl_2 + 3H_2O \tag{9}$$

[1]实际情况中，因为系数不为 0，所以设自由元为 1 最可能得到一般性的化学方程式

2.2 解析模块设计

但是，为了让用户输入的化学方程式转换为增广矩阵的形式。需要设计解析器 (Parser) 来对方程式进行解析。

设计的思路是：

1. 方程式按照等号拆成两边。
2. 按照 $+$ 拆为化学式。
3. 使用正则表达式对最内层的括号单位 (稍后介绍) 进行拆解
4. 重复 3 直到无法匹配到括号单位
5. 对单个处理过的化学式记录到表中
6. 记录完所有的化学式

这样，就完成了解析的工作。

2.2.1 正则表达式设计

括号单位，我们定义为形如这样的化学式：(t 为 1 时可省略)

$$(A_1c_1A_2c_2\cdots A_nc_n)_t \tag{10}$$

如 $(OH)_2$ 就属于括号单位。括号单位的正则表达式是：

```
1 \((([A–Z][a–z]*(\d+)*)+)\)(\d+)*
```

此外，为了在记录时便于获取各元素的系数，还可以使用如下正则表达式匹配元素与系数：

```
1 ([A–Z][a–z]*)(\d+)*
```

2.3 配平模块设计

配平模块主要是对原理的计算机实现，主要难点在于高斯-约当算法与如何实现“零精度误差”的分数数据结构。

2.3.1 高斯-约当消元的实现细节

直接实现的高斯-约当消元是极低效率且难以工作的。在实际实现中，需要注意以下两个特性细节的添加：

1. 对于当前行 (第 p 行)，与 $p+1$ 到 n 行中主元[2](pivot) 最左的行交换。
2. $p+1$ 到 n 行，交换后的当前行主元所在列中，选择绝对值最大的行交换。

2.3.2 分数数据结构细节

需要实现的有基本的加减乘除运算以及化简，返回实数，比较大小，绝对值等。

3 代码实现

下面所有代码已被折行，标准格式请在GitHub上查看。

[2]当前行第一个非零元素

3.1 Delta-3 代码核心解析

将表 (table) 作为对象来实现：

```
pub struct TableDesc {
    elements_table: HashMap<String, usize>, // 元素哈希检索表
    list: Vec<Vec<i32>>, // 存储表
    formula_sum: i32, // 化学式个数
}
```

impl 中的核心方法 (method) 是 store_in_table：

```
pub fn store_in_table(
  &mut self,
  formula: &str,
  location: usize
) -> Result<bool, ErrorCases> {
    for t in get_token(formula)? {
        if !self.elements_table
          .contains_key(&t.token_name) {
            // 检查元素是否已经出现
            let len = self.elements_table.len();
            self.elements_table.insert(
                t.token_name.clone(),
                len + 1,
            );
            self.update_list_vec();
        }

        {
            // 向表中写入数据
            let tmp = match self.elements_table.get(
                &t.token_name
            ) {
                Some(s) => *s,
                None => return Err(NotFound),
            };
            self.list[tmp][location]
              = match self.list[tmp][location]
                .checked_add(t.times) {
                Some(s) => s,
                None => return Err(I32Overflow),
            }
        }
    }
    Ok(true)
}
```

此外就是对括号进行拆解的功能 (function)：

```
fn parser_formula(
  // 解析化学式
  formula: &str,
  table: &mut TableDesc,
```

```
        location: usize,
    ) -> Result<bool, ErrorCases> {
        let formula_backup = formula;
        let mut formula = format!("({})", formula_backup);
        // 对于方程式左右加上"(",")", 使其满足括号单位的定义

        formula_spliter(&formula)?;
        while formula_spliter(&formula).is_ok() {
            for p in formula_spliter(&formula)? {
                // 每次拆分最内层的括号单位并替换
                formula = replace_phrase(
                    &formula,
                    &p.all,
                    &(mul_phrase(&p)?)
                );
            }
        }
        table.store_in_table(&formula, location)?;
        Ok(true)
    }
```

3.2 AlphaForce 代码核心解析

分数数据结构的实现就不在此赘述，只需要模拟即可，注意需要对每一步运算做溢出检测。

3.2.1 高斯-约当消元算法实现

这里介绍两个核心的结构设计：

```
        fn get_leftmost_row(&self, row: usize) -> Option<usize> {
        let mut fake_zero = false; //  “零锁”设计
        let mut leftmost = row;
        let mut min_left: usize = match self.get_pivot(row) {
            Some(s) => s,
            None => {
                fake_zero = true;
                0
            }
        };
        for i in row + 1..self.n {
            let current_pivot = match self.get_pivot(i) {
                Some(s) => s,
                None => continue, // 有全0行就跳过
            };
            if (current_pivot < min_left) | (fake_zero) {
                // 只要fake_zero为true就会替换，实现了零锁行最大
                leftmost = i;
                min_left = current_pivot;
                fake_zero = false;
            }
```

```
            }
            if fake_zero {
                None // 如果零锁仍存在说明全部为全0行
            } else {
                Some(leftmost)
            }
        }
```

零锁设计保证了代码在 $p+1$ 到 n 行全为 0 时有返回 None 而当任何一行不为全 0 时替换。
其余的部分就是对高斯-约当算法的实现，只需要学习线性代数即可。这里是实现：

```
        pub fn solve(&mut self) -> Result<
            ResultHandler<Vec<Frac>>,
            ErrorCases
        > {
            // The Gaussian–Jordan Algorithm
            for i in 0..self.n {
                let leftmosti = match self.get_leftmost_row(i) {
                    Some(s) => s,
                    None => continue,
                };
                self.matrix_a.swap(i, leftmosti);
                self.matrix_b.swap(i, leftmosti);
                let j = match self.get_pivot(i) {
                    // 如果“最左”行依旧为0，跳过
                    Some(s) => s,
                    None => continue,
                };
                let maxi = self.get_max_abs_row(i, j)?;
                if self.matrix_a[maxi][j].numerator != 0 {
                    self.matrix_a.swap(i, maxi);
                    self.matrix_b.swap(i, maxi); // 交换绝对值大的行
                    {
                        let tmp = self.matrix_a[i][j];
                        self.divide_row(i, tmp)?;
                    }
                    for u in i + 1..self.n {
                        let v = self.mul_row(i, self.matrix_a[u][j])?;
                        for (k, item) in v.iter().enumerate()
                            .take(self.m) {
                                self.matrix_a[u][k]
                                    = self.matrix_a[u][k].sub(*item)?;
                        }
                        self.matrix_b[u]
                            = self.matrix_b[u].sub(v[self.m])?;
                    }
                }
            } // 行梯阵式(REF)

            for i in (0..self.n).rev() {
                let j = match self.get_pivot(i) {
```

```
                    Some(s) => s,
                    None => continue,
                };
                for u in (0..i).rev() {
                    // j above i
                    let v = self
                      .mul_row(i, self.matrix_a[u][j])?;
                    for (k, item) in v.iter().enumerate()
                      .take(self.m) {
                        self.matrix_a[u][k] = self.matrix_a[u][k]
                          .sub(*item)?;
                    }
                    self.matrix_b[u] = self.matrix_b[u].sub(v[self.m])?;
                }
            } // 简化行梯阵式(RREF)
            let mut ans: Vec<Frac> = vec![Frac::new(0, 1); self.m];
            let pivots = self.check()?;
            let mut free_variable = false;
            for i in (0..self.m).rev() {
                if pivots.contains_key(&i) {
                    let mut sum = Frac::new(0, 1);
                    for (k, item) in ans.iter().enumerate()
                      .take(self.m).skip(i + 1) {
                        sum = sum.add(
                            self.matrix_a[pivots[&i]][k].mul(*item)?
                          )?;
                    }
                    ans[i] = self.matrix_b[pivots[&i]]
                        .sub(sum)?
                        .div(self.matrix_a[pivots[&i]][i])?;
                } else {
                    free_variable = true;
                    ans[i] = Frac::new(1, 1); // 设所有的自由元为1
                }
            }
            Ok(ResultHandler {
                warn_message: if free_variable {
                    FreeVariablesDetected
                } else {
                    NoWarn
                },
                result: ans,
            })
        }
```

4 总结

lib_xch 实现了化学方程式的配平基本需求，但是还不支持离子方程式等。离子方程式的配平原理与此也相同。此方法适用于大部分的方程式，但是还有部分缺陷, 如无法配平此方程：

$$KClO_3 + HCl = KCl + ClO_2 + Cl_2 + H_2O \tag{11}$$

使用 ILP 模型[3]可以配平：

$$2KClO_3 + 4HCl = 2KCl + 2ClO_2 + Cl_2 + 2H_2O \tag{12}$$

此方法还可以用于猜测新的化学方程式 (使用已知的线性无关解来线性表示)，等等。

库 (Library) 运用了一些 Rust 的特性，在我编写的过程中也真切地体会到 Rust 的用户友好 (user-friendly)，不但无需担心一些内存的分配与回收，而且还能在没有运行时 (runtime) 的情况下实现完整的异常处理 (使用 Result 的可恢复错误机制)。语言本身还结合了面向对象编程 (Object-Obriented Programming) 与函数式编程的特点 (如 Monad,Lambda 等)，是一门真正现代设计的语言。

与此同时 Rust 还天生提供了媲美 C 的运行速度以及优化，让库的运行效率提高不少。

但是，Rust 也有其缺点，如：编译效率低，生态不完全 (尽管现在已经发展了不少)，许多特性依赖编译等。不过，这对我而言算不了什么，因此，我选择了 Rust。的确，这也是一个正确的选择。

任何 BUG，意见或建议欢迎发邮件至LEXUGEyky@outlook.com或在 GitHub 上开issue。

[3]ILP 模型属于 NP 完全问题，时间复杂度不可预估，所以没有采用

References

[1] Blakley, G. R. (1982). Chemical equation balancing. Journal of Chemical Education, 59.

[2] Sen, S. K., Agarwal, H., & Sen, S. (2006). Chemical equation balancing: An integer programming approach. Elsevier Science Publishers B. V.