并行程序设计课设实验报告

学号：xxx　姓名：tky

北京航空航天大学软件学院

日 期：June 29, 2021

目录

1 实现介绍

在第一章中主要是对实现的介绍。本章的思路组织如下：

- 在**第 1.1 节**、**第 1.2 节**中介绍遇到的问题与解决措施。
- 在**第 1.3 节**中介绍如何将任务进行并行化与分块。
- 在**第 1.4 节**中介绍如何提速、增强程序的扩展性。

1.1 基本问题与措施

本小节讨论一些基础问题的措施。

本课设针对的问题是多义性文法的异构语法树个数判定问题。这本身是一个经典的动态规划（Dynamic Programming, DP）问题，于上个世纪就由 John Cocke, Daniel Younger, Tadao Kasami 三位发现者提出了 CYK 算法解决 [1]。算法的伪代码描述如下：

```
let the input be a string I consisting of n characters: a1 ... an.
let the grammar contain r nonterminal symbols R1 ... Rr, with start symbol R1.
let P[n,n,r] be an array of booleans. Initialize all elements of P to false.

for each s = 1 to n
    for each unit production Rv  as
        set P[1,s,v] = true

for each l = 2 to n -- Length of span
    for each s = 1 to n-l+1 -- Start of span
        for each p = 1 to l-1 -- Partition of span
            for each production Ra     Rb Rc
                if P[p,s,b] and P[l-p,s+p,c] then set P[l,s,a] = true

if P[n,1,1] is true then
    I is member of language
else
    I is not member of language
```

宏观来看，整个 DP 流程最关键的状态转移过程是“将长度为 l 的大序列切分 $l-1$ 次（枚举左右非空连续子串），循环进行检查匹配，若与产生式匹配成功则标记（计数）”。基于这一基本算法，我们可以设计整个问题求解程序。编写程序的几个主要重点问题与解决措施如下：

1.1.1 数据读入

按照作业规定的文件进行读入即可。由于 IO 数据量相比计算量很小，因此此处没有太大必要进行读入优化。

1.1.2 整体算法实现

伪码描述中对最内层循环的产生式匹配检查不够具体，因而该部分将是一个重要的需要人为思考实现方式的地方。实际上该处有两种实现方式：

(1) **枚举产生式进行循环匹配**：遍历所有产生式，若发现一个产生式<a>::=<b><c>满足：

- 当前枚举的左子串**可以**规约为非终结符 <b>
- 当前枚举的右子串**可以**规约为非终结符 <c>

则当前整个子串**可以**通过该产生式规约为非终结符 <a>，于是计数即可。

(2) **成对枚举左、右子串各自能够规约成的非终结符集合**：换言之是枚举左右子串能规约成的非终结符的笛卡尔积 [<b>, <c>]，随后查询全部右侧为<b><c>的产生式，计数即可。

由于本小节主要关注基本的整体程序实现，因此关于两种实现方式和各自的主要问题、可能的解决方案将于后文（**第 1.2 节**）讨论。

1.1.3 数据输出

按照作业规定，输出到标准输出即可。由于 IO 数据量相比计算量很小，因此此处没有太大必要进行输出优化。

1.2 核心算法问题与措施

本小节主要是对**第 1.1.2 节**中提到的两种实现思路各自的核心算法问题和解决方案做探讨。

1.2.1 思路一：枚举产生式

(1) **主要问题**：是如何提高最内层循环（也就是这个枚举产生式的循环）的枚举效率。对于思路 2，待枚举的笛卡尔积是动态的（和左右子串能推出的非终结符集合的大小有关），因此一定程度上可以压低时间复杂度的常数项。而对于思路 1，如果不进行任何优化，复杂度将是一个**紧的**大常数上界。

(2) **可能的解决方案**：是对外部的两个枚举变量（子串的左右指针，记为 l, r）进行记忆化，即每当遍历完成某个 l, r 对并匹配成功时，将对应的记忆化数组位置进行标记。于是在枚举第三层循环的变量 $m \in [l, r]$ 时，一旦发现 l, m 或 m, r 对应位置的记忆化数组从未被标记，则说明这个子串无法推出任何非终结符，即可跳过本次对产生式的枚举，达到降低时间复杂度常数项的目的。

1.2.2 思路二：枚举可规约非终结符的集合的笛卡尔积

(1) **问题主要转化为**：如何快速维护每个连续子串所能规约成的非终结符集合。为了讨论方便，后文记每次每个集合的元素个数为 N。

(2) **措施 1**：是范例串行代码提供的一个基本思路，即使用插入排序的方式维护一个升序集合。这种实现方式利用的是一**线性数据结构**对集合进行数据结构功能上的模拟（互异性），因此为了维护集合，每次插入的时间复杂度粗略为 $O(N)$ 的，速度较慢。

(3) **措施 2**：考虑使用**树状数据结构**，例如平衡树。实际这里可以使用标准模板库（STL）提供的`std::set`实现。`std::set` 底层为红黑树，故插入查询复杂度是一个小常数的 $O(\log N)$.

(4) **措施 3**：考虑使用**散列数据结构**，例如哈希表。实际这里可以使用标准模板库（STL）提供的`std::unordered_set`，或者扩展标准库的`pbds::gp_hash_table` 实现，插入和查询的复杂度（均摊）都降到了 $O(1)$.

(5) **措施 4**：考虑使用**状态压缩**，利用一个 128bits 的数据结构即可维护这个集合。实际可以选用`__int128`或者`std::bitset<128>` 来实现，插入复杂度是很低的常数 $O(1)$，查询需要逐个比特枚举，是 $\Theta(128)$ 但由于是位运算因此开销也很小。

1.2.3 最终思路

最终考虑到实现和计算的简洁性，选择了思路一。

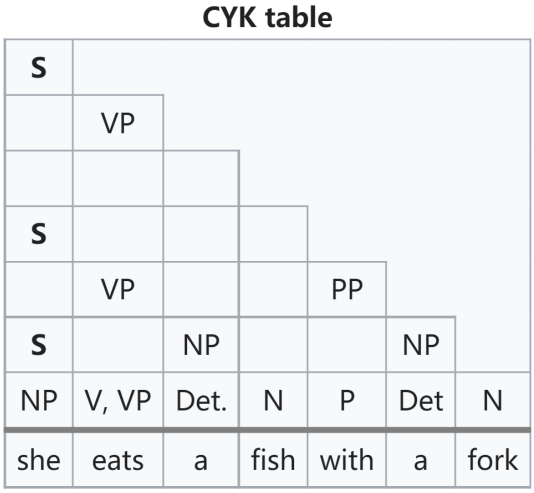

图 1: CYK 算法解析某个句子的示意图。可以看到 CYK 使用的动态规划表可以理解为在自底向上地、辗转循环地遍历一个三角矩阵。

1.3 并行化与任务分块

本小节讨论并行化和任务的分配。整个 CYK 算法的工作流程示例类似**图 1** 所示。CYK 对动态规划表的填充是自底向上（假设最左下角为表的起点）、辗转循环的。这样的填充顺序是**存在依赖的**，且破坏这种依赖会导致算法正确性的错误。

为了在满足正确性的同时将任务并行化，我们可以采取如下的任务分块方式，并在不违反数据依赖性的前提下进行并行化：

1.3.1 分块和并行策略

我们考虑将最小粒度的任务定义为“确定当前待推导子串后，逐个枚举子串的切分位置，并进行产生式匹配”。确定这个最小粒度的任务需要两个量，即当前待推导子串的左右界（记为 l, r）。我们约定使用 $T_{l,r}$ 表示一个分块的子任务，即对子串 $S[l:r]$ 进行枚举切分位置 m，并进行产生式的匹配。

考虑**数据计算的依赖关系**，每次计算 $T_{l,r}$ 时，需要一个子任务集合 $\mathcal{P}_{l,r}$ 中的任务全部计算完毕，其中 $\mathcal{P}_{l,r}$ 是满足如下条件的子任务集合：

$$\mathcal{P}_{l,r} = \{T_{l,m} \mid \forall m, l < m < r\} \cup \{T_{m,r} \mid \forall m, l < m < r\} \tag{1}$$

因此如果我们直接让线程随意地在全部子任务 $T_{l,r}$ 中选取，必定会因为违反数据计算依赖导致结果错误。为了在并行时保证算法正确性，我们需要加上约束，即**若该线程需要访问某个子任务，需要等待该子任务对应的前置集合完成**。

实际观察**图 1** 所示的动态规划方式，我们不难发现，每个子任务的计算结果在动态规划表中**只会写入一次，同时会读出多次**。因此该处类似“接力”的同步，可以通过忙等实现：当线程遇到需要等待前置集合完成时，反复循环读取被`volatile` 标记的某个全局变量地址，直到该地址被某线程写入标记完成后，再停止忙等，继续执行子线程。此处不会发生写冲突，因为按照算法流程，每个地址均只会被写入一次。

1.3.2 所选策略的优势

这样的子任务划分和并行策略有如下好处：

(1) **更高效的同步方式**：相比**串行执行最外层**循环的方式（即在第一层循环的每次开始时，使用类似`barrier`的方式同步全部线程的粗暴同步方式），这种**“部分并行”**最外层循环的方式可以节省大量时间。

(2) **更高效的任务调度**：可以发现这种实现方式能够配套使用**动态调度**，即使用一个**原子变量**标记当前待分配的子任务（最外层循环变量），并让每个线程逐个领取对应的子任务，**而不是**在算法执行前就静态地决定好每个线程的全部子任务，达到了更高的效率和更好的线程负载均衡。

1.4 提速与扩展性

本小节讨论如何加速程序，以及如何保证扩展性。

1.4.1 如何提速

在合理选择了如**第 1.3.1 节**所介绍的并行策略后，已经可以使用动态调度，整体程序的速度和并行度已经达到了一个很可观的水平。

而为了追求更极致的速度，还有一些额外的优化程序速度的方法列举如下：

(1) **提高局部性**：提高程序对内存访问的局部性是课程所介绍的一个重大优化思路，其主要优势在于能够帮助缓存命中。程序中对产生式进行了排序，提高了访问动态规划数组的局部性。虽然对产生式是否进行排序**不会影响程序的正确性**（因为选择了循环产生式的思路，而非集合合并的思路，因此二元产生式的顺序不影响计算结果），但最终代码中仍然选择对产生式进行了**按产生式中三个编号的升序**依次排序，这样可以在动态规划最内层遍历产生式时，保证对相邻的两个产生式各自指向的动态规划数组访问时，访存尽可能接近。

(2) **尽可能使用栈变量**：若一些变量（尤其是数组变量）不太大，则可以尽量开辟在栈空间，而不是使用全局变量，更不必通过动态内存分配开辟。由于程序总是在高频访问栈空间，整体上而言对栈内存的访问更局部、更快。且栈变量的地址可直接通过寄存器和立即数得到，访存开销更小。

(3) **产生式剪枝**：考虑全体产生式构成了一个从 0 结点出发的有向图。因此我们可以从 0 结点开始对可达性进行分析（一个简单的广度优先或深度优先搜索即可；非递归的广度优先结合邻接表或者链式前向星的数据结构往往可以得到更低的时间复杂度常数），并直接舍弃不可达的产生式，达到剪枝的目的。

(4) **辅助编译器进行优化**：主要是合理地使用宏定义、宏函数、`constexpr`、`noexcept` 等可以帮助编译器优化的工具。

(5) **利用好 O3 的编译指令，合理使用 STL**：在 O3 优化下，标准模板库的许多数据结构和函数是非常快的，因此没有必要进行手写排序函数等优化，反而很可能会导致负向优化。

(6) **利用好 C++11 的编译指令，减少不必要的开销**：C++11 的一些特性，例如右值引用和移动语义，可以减少对大内存对象的不必要拷贝。当然合理使用指针和左值引用也可以实现低拷贝开销（实际只是多一个指针变量的开销）。

1.4.2 如何保证程序的扩展性

为了保证程序的扩展性，程序进行了如下的设计：

(1) **使用动态线程任务分配而不是静态**：如**第 1.3.1 节**所介绍的并行策略的并行方式，使用一个简单的原子整型变量实现动态的任务领取，即可在线程调度和计算之间取得一个动态平衡，提升不同线程个数下的表现。

(2) **利用宏判断是否开启多线程**：许多程序都提供了一个简单的宏定义或常量定义表示线程个数，但如果直接将该数修改为 1，某些程序可能会崩溃。在个人编写代码时**额外注意**了这一点，利用宏在编译期的判断语句应对了串行和并行的情况，并进行了不同的代码编译，提高了程序扩展性的同时也精简了串行代码。

2 分析与实验

本章节主要是分析对**程序运行时间的测定**，及进行**可视化**、和**原因分析**。

2.1 时间测试说明

在本课设中，个人使用了高性能计算集群的计算结点进行测速。测试使用了`srun` 调度系统提交测速脚本，脚本用`Python` 语言编写。测试脚本编写思路如下：

(1) 自动遍历路径下的全部`*.c` 或`*.cpp` 文件，进行编译。

(2) 使用子进程调用类`unix` 系统的`time` 指令进行测速。

(3) 通过管道读回输出在标准错误的测试结果，并解析。

(4) 多次使用指令测速并将解析的结果求中位数。

(5) 整理得到最终结果。

详细的代码可见**第 4 节**。

2.2 工具说明

本课设使用了`pthread` 进行多线程编程，没有使用`OpenMP` 或`MPI`。

2.3 比较图

根据前述时间测试方式，个人对最终提交的代码在集群上进行了测速。测速时将程序代码中的线程数从 1 变化到了 36，进行了多组测试（每次测速时均在具备 36 个核的机器上运行编译完成的程序）。后续两个小节将对不同的比较图进行分析（使用 Excel 工具作图）。

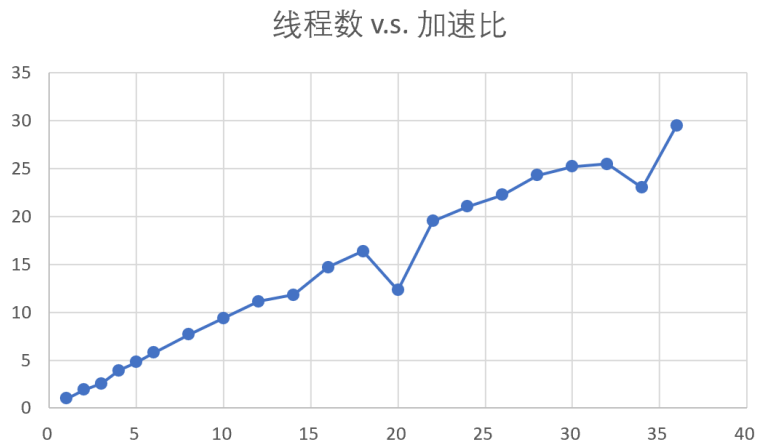

图 2: 加速比比较图。横轴是代码中的实际的多线程个数；纵轴是加速比。除了少数波动外，基本可以观察到线性加速比，说明程序的并行化非常成功。

2.3.1 加速比比较图和原因分析

加速比比较图和横纵轴情况如**图 2** 所示，加速比（纵轴）关于线程个数（横轴）基本成**线性关系**。

分析原因：能够实现线性加速比说明了程序使用的并行策略的效果很好。这主要由于使用了动态分配线程任务的方式。关于更详细的并行策略和原因分析，请见**第 1.3 节**。

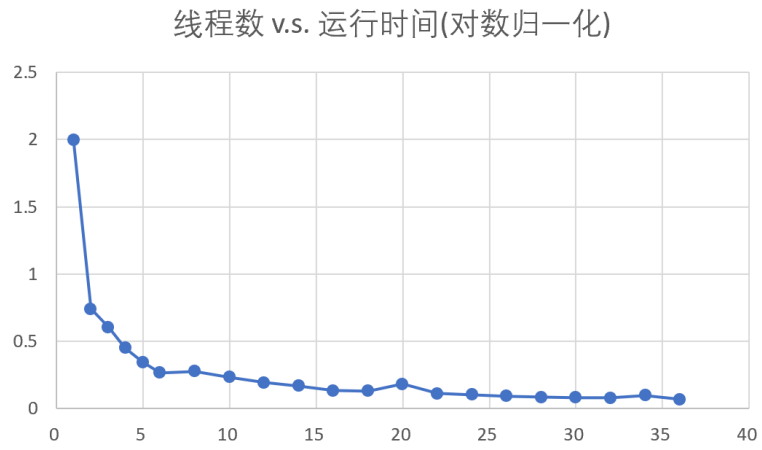

图 3: 执行时间比较图。横轴是代码中的实际的多线程个数；纵轴是三组数据运行时间的几何平均（对运行时间的乘积使用了对数归一化，即将乘积 x 转为 $\log(1+x)$ 作为纵轴的取值）。

2.3.2 执行时间比较图和原因分析

执行时间比较图和横纵轴情况如**图 3** 所示，也展现出了多线程加速的成功（随着线程个数增加，运行速度明显加快多个数量级）。由于使用了对数归一化，实际上相比纯串行的加速倍数已经达到 10^2 以上的数量级。

分析原因：执行速度达到了很高的水平，除了实现线性加速比和成功并行化以外，与**第 1.4 节**中介绍的各种加速技巧也有很强的因果关系。有关具体加速技巧、和为什么能够加速的原因分析，请详见**第 1.4 节**。

3 结论

本程序的最佳方案由三部分组成：

(1) 选用了合适的核心算法（具体优势请见**第 1.2.3 节**介绍；主要优势是计算简洁、时间复杂度友好）。

(2) 选用了合适的并行策略（具体优势请见**第 1.3 节**介绍；主要优势是可做到动态的线程任务调度）。

(3) 使用了大量的代码优化技巧，保证高效率和高扩展性（具体优势请见**第 1.4.1 节**介绍；主要优势是利用好 C/C++ 的各种特性和工具）。

4 附加项一：提出建议与解决方案

4.1 问题建议

个人在公司服务器上测试运行速度时，发现存在一定波动的情况。为了使最终的评测更为准确，个人提出了“多次测速，并使用受极端值影响较小的**中位数**作为最终评价指标”的建议。

与助教沟通后，助教采纳了这一建议。个人也提供了一个脚本作为解决方案。

4.2 提供的解决方案：自动评测脚本

详细的脚本设计思路请见**第 2.1 节**。完整的脚本代码如下，关键之处配有对应注释进行解释：

```python
import os
import datetime
import shutil
import subprocess
import sys
import time
from collections import defaultdict, OrderedDict
from copy import deepcopy
from random import randint
from subprocess import TimeoutExpired
from pprint import pformat

import numpy as np
import pandas as pd
from tqdm import tqdm


# 获取当前时间描述字符串，主要是为了对 log 文件命名
def time_str():
    return datetime.datetime.now().strftime('%m%d_%H-%M-%S')

# 获取中位数
def get_mid(ls: list):
    ls = sorted(ls)
    mid = len(ls) // 2
    if len(ls) % 2 == 0:
        return (float(ls[mid]) + float(ls[mid - 1])) / 2
    else:
        return float(ls[mid])
```

```
30
31  # 解析 time 指令的输出，获得墙上时间、用户态时间、核心态时间
32  def get_secs(stderr_file_content: bytes):
33      time_strs = list(filter(lambda s: len(s) > 3,
34                  stderr_file_content.decode('unicode_escape').splitlines()))[-3:]
35
36      def to_secs(time_str: str):
37          prefix, time_str = time_str.split()
38          m, s = time_str.split('m')
39          s = s.strip('s')
40          secs = float(m) * 60 + float(s)
41          return secs
42
43      real_sec = to_secs(time_strs[0])
44      user_sec = to_secs(time_strs[1])
45      sys_sec = to_secs(time_strs[2])
46
47      return real_sec, user_sec, sys_sec
48
49
50  # 主函数，进行测速
51  def main():
52      # 得到当前文件夹下的全部待编译文件
53      c_cpp_files = sorted(list(filter(lambda f: os.path.splitext(f)[1] in {'.c', '.cpp'}, os.listdir())))
54      f2 = deepcopy(c_cpp_files)
55      import random
56      random.seed(round(time.time() * 10))
57      for _ in range(10):
58          random.shuffle(c_cpp_files)
59          random.shuffle(f2)
60      c_cpp_files = c_cpp_files
61      print(f'all c_cpp_files: {c_cpp_files}', file=sys.stderr, flush=True)
62
63      results = OrderedDict({
64          'name': [], 'status': [],
65          'ac1': [], 'real1': [],
66          'ac2': [], 'real2': [],
67          'ac3': [], 'real3': [],
68          'geo_mean': [],
69          'user1': [], 'sys1': [],
70          'user2': [], 'sys2': [],
71          'user3': [], 'sys3': [],
72          'real1_all': [], 'user1_all': [], 'sys1_all': [],
73          'real2_all': [], 'user2_all': [], 'sys2_all': [],
74          'real3_all': [], 'user3_all': [], 'sys3_all': [],
75      })
76      bar = tqdm(c_cpp_files)
77      for c_cpp_file in bar:
78          job_name = os.path.splitext(c_cpp_file)[0]
79          results['name'].append(job_name)
80          results['status'].append('RUN')
81
82          # 编译，如果编译出错则记录结果为 “CE”
```

```
        compile_cmd = f'g++ -std=c++11 -O3 -mavx2 -lpthread {c_cpp_file} -o a.out'
        compile_subproc = subprocess.Popen(compile_cmd, shell=True, stdout=subprocess.PIPE,
                          stderr=subprocess.PIPE, bufsize=-1)
        compile_subproc.wait()
        if compile_subproc.poll() == 0:
            pass  # 编译通过
        else:  # 编译错误
            compile_subproc.kill()
            results['status'][-1] = 'CE'
            for k, v in results.items():
                if k not in {'name', 'status'}:
                    v.append('CE')
            continue

        # 循环 N 次执行 time 指令测速
        # 只要有一次墙上超过 MAX_SECS 秒（下面定义成 300 了，也就是 5 分钟），则记为 TLE，并马上跳到下一组数据
        # 只要有一次运行时错误，则记为 RE，并马上跳到下一组数据
        N = 4
        MAX_SECS = 300

        for data_idx in range(1, 4):

            with open('input.txt', 'w') as finp:
                with open(f'input{data_idx}.txt', 'r') as finpx:
                    content = finpx.read().replace('\r\n', '\n').replace('\n', '\r\n')
                finp.write(content)
            # shutil.copy(f'input{data_idx}.txt', 'input.txt')

            ac = True
            real_secs, user_secs, sys_secs = [], [], []
            for n in range(N):
                bar.set_description(f'[{job_name}, data{data_idx}] [{n + 1}/{N}] ')
                time_subproc = subprocess.Popen('time ./a.out', shell=True, stdout=subprocess.PIPE,
                stderr=subprocess.PIPE, bufsize=-1)
                try:
                    time_subproc.wait(MAX_SECS)
                except subprocess.TimeoutExpired:
                    time_subproc.kill()
                    ac = False
                    results['status'][-1] = 'TLE'
                    real_secs.append('TLE')
                    user_secs.append('TLE')
                    sys_secs.append('TLE')
                    break

                if time_subproc.poll() == 0:
                    ans, time_ans = time_subproc.communicate()
                    ans = int(ans.decode('unicode_escape').strip())
                    with open(f'output{data_idx}.txt', 'r') as fgt:
                        ac &= int(fgt.readline().strip()) == ans
                        if not ac:
                            results['status'][-1] = 'WA'
                    real_sec, user_sec, sys_sec = get_secs(time_ans)
```

```
                    real_secs.append(f'{real_sec:.4g}')
                    user_secs.append(f'{user_sec:.4g}')
                    sys_secs.append(f'{sys_sec:.4g}')
                else:
                    time_subproc.kill()
                    ac = False
                    results['status'][-1] = 'RE'
                    real_secs.append('RE')
                    user_secs.append('RE')
                    sys_secs.append('RE')
                    break

                bar.set_postfix(OrderedDict({'last': real_secs[-1], 'ans': ans, 'ac': ac, 'sta': results['statu

            results[f'ac{data_idx}'].append('AC' if ac else results['status'][-1])
            results[f'real{data_idx}'].append(f'{get_mid(real_secs):.5g}' if ac else results['status'][-1])
            results[f'user{data_idx}'].append(f'{get_mid(user_secs):.5g}' if ac else results['status'][-1])
            results[f'sys{data_idx}'].append(f'{get_mid(sys_secs):.5g}' if ac else results['status'][-1])
            results[f'real{data_idx}_all'].append(str(np.array(sorted(real_secs))) if ac
            else results['status'][-1])
            results[f'user{data_idx}_all'].append(str(np.array(sorted(user_secs))) if ac
            else results['status'][-1])
            results[f'sys{data_idx}_all'].append(str(np.array(sorted(sys_secs))) if ac
            else results['status'][-1])

        # 计算三个墙钟时间中位数的几何平均
        try:
            t1 = float(results["real1"][-1])
            t2 = float(results["real2"][-1])
            t3 = float(results["real3"][-1])
            mean = f'{(t1 * t2 * t3) ** 0.5:.4g}'
        except:
            mean = 'not AC'

        results[f'geo_mean'].append(mean)

    del results['status']

    pd.set_option('display.max_rows', None), pd.set_option('display.max_columns', None)
    pd.set_option('max_colwidth', 100), pd.set_option('display.width', 200)
    less_results = deepcopy(results)
    for k in results.keys():
        if k.endswith('_all'):
            del less_results[k]
    print(pd.DataFrame(less_results).sort_values('geo_mean', ascending=True), file=sys.stderr, flush=True)

    df = pd.DataFrame(results).sort_values('geo_mean', ascending=True)
    fname = f'results-{time_str()}.csv'
    df.to_csv(fname, index=None)
    print(f'\n[finished!] log file has been saved at {fname}', file=sys.stderr, flush=True)


if __name__ == '__main__':
```

```
189     main()
```

5 附加项二：大数据生成

个人提供了一个大数据生成脚本如下：

```
1   import os
2   import sys
3   import time
4   import random
5   import datetime
6   from pprint import pprint as pp
7   from pprint import pformat as pf
8   import numpy as np
9
10
11  # 大数据生成脚本；由于主要是大数据，因此默认使用了最大的取值
12  def main():
13      vn_num = 128
14      production2_num = 512
15      production2 = []
16      production1 = []
17      string_length = 1024
18
19      production1_num, string = generate(vn_num, production2_num, production2, production1, string_length)
20
21      with open('input.txt', 'w') as fout:
22          print(vn_num, file=fout)
23
24          print(production2_num, file=fout)
25          for pa, ch1, ch2 in production2:
26              print(f'<{pa}>::=<{ch1}><{ch2}>', file=fout)
27
28          print(production1_num, file=fout)
29          for vn, vt in production1:
30              print(f'<{vn}>::={vt}', file=fout)
31
32          print(string_length, file=fout)
33          print(string, file=fout)
34
35
36  def generate(vn_num, production2_num, production2, production1, string_length):
37
38      # 使用全部非终结符，凑出基本的连接
39      vns = list(range(vn_num))
40      random.shuffle(vns)
41      if vn_num % 3 != 0:
42          extra_vns = random.choices(vns, k=vn_num % 3)
43          vns.extend(extra_vns)
```

```
44
45      for i in range(0, len(vns), 3):
46          j, k = i+1, i+2
47          production2.append((vns[i], vns[j], vns[k]))
48          production2_num -= 1
49
50      assert production2_num >= 0
51
52      # 进行密集连接
53      dense_vns = list(range(vn_num))
54      random.shuffle(dense_vns)
55      dense_vns = dense_vns[:max(round(len(dense_vns) * 0.08), 3)]
56      assert len(dense_vns) > 0
57
58      for i in range(production2_num):
59          if random.randrange(4) == 0:
60              pa = 0
61          else:
62              pa = random.choice(dense_vns)
63          ch1 = random.choice(dense_vns)
64          ch2 = random.choice(dense_vns)
65          production2.append((pa, ch1, ch2))
66
67      # 产生字符串和 production1
68      production1_num = len(dense_vns)
69      for v in dense_vns:
70          production1.append((v, random.choice([chr(ord('a') + x) for x in range(26)]) ))
71
72      alphas = set([p[1] for p in production1])
73
74      random.shuffle(production2)
75      random.shuffle(production1)
76
77      return production1_num, random.choices(alphas, k=string_length)
78
79
80  if __name__ == '__main__':
81      main()
```

参考文献

[1] GONZALEZ R C, THOMASON M G. Syntactic pattern recognition: An introduction[J]. 1978.

[2] TAKASHI N, KENTARO T, TAURA K, et al. A parallel cky parsing algorithm on large-scale distributed-memory parallel machines[J]. 1997.