### GCTA风云挑战赛--FASTQ数据的高倍数压缩算法的说明文档

Rong-Hua He

2006-Oct

**摘要：**对于GCTA风云挑战赛--FASTQ所描述的问题，我针对文档中的三类数据分别进行了初步压缩。这三类数据包括相关描述信息、碱基信息和质量值信息。对于描述信息，如果将其顺次排列，每一条信息和紧随其后的信息之间的联系是十分紧密的，先后排列的序号或者一些类似于仪器编号的数字或字符的规律性很强，这些特性可以利用于压缩。对于碱基信息，我利用碱基只有四种的这种基本特性，将ACGT四种字符，直接转换为二进制码00,01,10,11以对其进行压缩。对于质量值信息，原文件用ASCII码表述其值，我利用其只包含40种不同值和值的分布特性，将其仅用4位二进制码来表示进而压缩。根据以上技巧，可以将这类FASTQ信息在直接可读取的条件下，初步压缩率约为29%。值得注意的是，这种格式的数据可以直接顺次读取，也就是说，可以在不经过解压的条件下，直接按顺序读出原始数据。这种数据可以经过成熟的软件进行再次压缩。但是令人遗憾的是，经过初步压缩后的数据，在rar或tar压缩格式情况下，仅能被压缩至原始数据的15%。此文档包括以下几个部分：（1）数据特性与压缩方法，（2）使用方法，（3）压缩效果。

1. **数据特性与压缩方法**

原始数据为ASCII码，在不同类别的信息中，包含着不同的规律性。有些信息虽然表面上是ASCII码，但是实际上只是使用了ASCII码中的小部分。我们知道一个ASCII码需要8个二进制码来表示，8个二进制码一共可以表示28 = 256种不同的情况，而我们可见的ASCII码仅仅是从其中的第32个（空格符号）到第126个（~符号）这95种情况（见附录3），其余的大约60%的无法显示的字符都并没有被利用。而即使是这95种可见的字符，也远远没有被完全利用。另一方面，对于描述信息来说，比如说实验的编号，仪器的名称等信息多数是重复的，或者有着明显的先后顺序，这也是可以被利用的。在这一章我们将通过三个小节分别阐述了描述信息、碱基信息和质量信息的数据特性，并给出了相应的压缩方法。

**1.1 描述信息**

描述信息指的是每一个基因序列单元的第一行，以’@’开头，表示的是序列的开头以及相关的描述信息。这里为了描述这类信息的特征，我们把NA12878\_1.fastq的前10行列出如下：

@ERR194147.1 HSQ1004:134:C0D8DACXX:1:1104:3874:86238/1

@ERR194147.2 HSQ1004:134:C0D8DACXX:2:2104:2852:75174/1

@ERR194147.3 HSQ1004:134:C0D8DACXX:3:1101:1318:114841/1

@ERR194147.4 HSQ1004:134:C0D8DACXX:3:1202:21337:51635/1

@ERR194147.5 HSQ1004:134:C0D8DACXX:3:2101:20539:23194/1

@ERR194147.6 HSQ1004:134:C0D8DACXX:3:2208:18292:174203/1

@ERR194147.7 HSQ1004:134:C0D8DACXX:4:1102:21352:12372/1

@ERR194147.8 HSQ1004:134:C0D8DACXX:4:1104:17663:42919/1

@ERR194147.9 HSQ1004:134:C0D8DACXX:4:1204:13406:85431/1

@ERR194147.10 HSQ1004:134:C0D8DACXX:4:2205:3830:6608/1

这里的每一行的字符大致可以分为三类，分别是：

1. 每一行都重复的内容（我们把它们标记为蓝色比如说@ERR，HSQ1004:），
2. 连续两行为等差数列的数字（我们把它们标记为红色或者绿色），
3. 并没有明确规律的数字（我们把它们标记为黑色）

我们把它们分别存储在不同的文件中。对于一个大体上保持不变的字符串，比如说@ERR，我们只需要记录一个@ERR，再接下来记录在每一行同一个位置反复出现的次数就可以了。这种情况下我们需要注意的是，也许在某一行文件内容发生了改变，因此在重复次数结束后，我们将记录新的字符串，比如说其改变成了@EQQ。这样导致我们记录的内容只有那些在这个位置连续出现的字符串的样子（若干个字符串）和其连续重复出现的次数（与字符串相同个数的正整数）。

对于那些看上去很可能是等差数列的内容，比如说1, 2, 3, 4, 5, 6, 7, 8, 9, 10或者134, 134, 134, 134, 134,……我们只需要记录其首位和公差就可以了。当然，这种等差的效果并不总是这样的，有可能在某一行公差发生了变化，因此，我们需要记录满足这种公差规律的行数，当公差变化时，再继续记录下一组。这样导致我们只需要记录首项、公差、重复次数这三个为一组的数量。而当公差发生变化的时候，我们只需要再记录这样三个数就可以了。

值得一提的是，无论我们是否才用ASCII码来存储或者是用二进制来存储，以上这些文件都非常小，特别是数据量大时，这种优势更加明显。当然，在我的程序中，为了让其尽可能小，都是采用二进制来存储。

根据上面所说的，我们只需要记录那些看上去并没有什么规律的那部分数就可以了，也就是那些黑色字体的数。这样就已经大大减少了所需要存储的内容。而且我们在记录这些数据的时候，采用二进制来记录，将大幅度减少所占用的空间。

**1.2碱基**

我们把前十行的碱基信息列举如下：

-----------------------------------------------------------------------------------

GGTTCCTACTTCAGGGTCATAAAGCCTAAATAGCCCACACGTTCCCCTTAAATAAGACATCACGATGGATCACAGGTCTATCACCCTATTAACCACTCACG

ACTTCAGGGTCATAAAGCCTAAATAGCCCACACGTTCCCCTTAAATAAGACATCACGATGGATCACAGGTCTATCACCCTATTAACCACTCACGGGAGCTC

CCAGCGTCTCGCAATGCTATCGCGTGCATACCCCCCAGACGAAAATACCAAATGCATGGAGAGCTCCCGTGAGTGGTTAATAGGGTGATAGACCTGTGATC

CAGCGTCTCGCAATGCTATCGCGTGCATACCCCCCAGACGAAAATACCAAATGCATGGAGAGCTCCCGTGTGTGGTTAATAGGGTGATAGACCTGTGATCC

AAAACCCAATCCACATCAAAACCCCCTCCCCATGCTTACAAGCAAGTACAGCAATCAACCCTCAACTATCACACATCAACTGCAACTCCAAAGCCACCCCT

CCAGCGTCTCGCAATGCTATCGCGTGCATACCCCCCAGACGAAAATCCCAAATGCCTGGAGAGCTCCCGTGAGTGGTTAATAGGGTGATAGACCTGTGATC

CCAGCGTCTCGCAATGCTATCGCGTGCATACCCCCCAGACGAAAATACCAAATGCATGGAGAGCTCCCGTGAGTGGTTAATAGGGTGATAGACCTGTGATC

CCAGCGTCTCGCAATGCTATCGCGTGCATACCCCCCAGACGAAAATACCAAATGCATGGAGAGCTCCCGTGAGTGGTTAATAGGGTGATAGACCTGTGATC

CCAGCGTCTCGCAATGCTATCGCGTGCATACCCCCCAGACGAAAATACCAAATGCATGGAGAGCTCCCGTGAGTGGTTAATAGGGTGATAGACCTGTGATC

TTAAATAAGACATCACGATGGATCACAGGTCTATCACCCTATTAACCACTCACGGGAGCTCTCCATGCATTTGGTATTTTCGTCTGGGGGGTATGCACGCG

-----------------------------------------------------------------------------------

对于碱基信息，正如赛题说明中所说，由A，C，G，T四种碱基构成，N为测量失败时的替补码。再根据对原始数据的观察，不难发现它至少具有以下三种特征：

1. 只有四种碱基。而却都以ASCII码来存储记录。
2. 大部分的结果都是由4种碱基构成的，而N只是个例外的情况。
3. 有时，在连续的若干的read单元中，碱基序列是不变的。我猜测这可能是对同一个基因序列或类似的基因的多次反复测量。（如碱基举例中的红色部分）

针对这三种情况，我结合了三种手段对碱基序列进行压缩，分别把结果以二进制的形式记录在若干个文件。

首先因为只有4种碱基，2的平方恰好为4，也就是说，我们可以利用2个二进制数就可以成功地表示出4种碱基。而原来用的A, C, G, T这种字符串形式，每个碱基对将占用一个字符，也就是8个二进制数。但就这一点来说，如果我们用00, 01, 10, 11来表示四种碱基，将会把数据压缩为原来的2/8，即25%。这部分内容我们记录在第一个文件中，文件名为bp\_0.bin（bp即为base-pair）。

其次，而对于那个替补码N来说，我们发现N的数量是很少的，多数的read单元都不包含N。因此，我们只需要记录出现的N的read单元的序号。再记录相应的N出现的位置信息就可以。而在碱基序列中，我们可以把N出现的位置都以某一种形式代替。在我的程序中，均以代表T的二进制码11来代替。

最后，对于那些重复的整个的碱基序列，如碱基举例中标红的那部分。我们只需要记录重复是从第几个单元开始的，以及重复的次数就可以。而在bp\_0.bin文件中，我们只需要记录重复开始的那个read单元的碱基结果就可以了。

通过以上的三种手段，我们基本可以把原来的数据压缩到大约25%，因为其中N的比例是很小的。这里值得一提的是，我们没有把连续重复的内容进行压缩（蓝色的部分），原因是：比如我们把GGGGG记为G5，那么我们就要用一个标记来表明此处出现了一个连续，而连续的数字需要用一个二进制码来表示，这就大大增加了记录所要的空间，而且重复的相同的碱基部分往往没有超过5个，而记录一个标记以及一个正整数就远远超过了4个碱基所占用的位置。这就导致了这种压缩表面上G5短于GGGGG，而实际上并不实用。

**1.3测量质量值**

这里列举了前十行的碱基序列值：

-----------------------------------------------------------------------------------

CC@FFFFFHHHHHJJJFHIIJJJJJJIHJIIJJJJJJJJIIGIJJIJJJIJJJIJIJJJJJJJJJJIJHHHHFFFDEEEEEEEEDDDCDDEEDDDDDDDDD

@BBFDFFFHFFHHIHIJJJJFIHHFHFHJCIHFHIJJJJJJJIJIJIJJIIHIJJJJJJJBEGIGHIHGHHHEFCDFFEDEEDEEDDD?CCCDDDDDDDDC

@@@FFFFFHHFFHGJJCGGHHIIGFHHGIGIF>GHIJJIJJFHGFCDD@CCDECDDEDD:<:<?CCABDAB8?::<@<>@ACCDD9?BDECCCDDDCDCCD

@@@ADDDDBFA8DAGBF4CAC1:E?CFFEBBBEF@05(-;;8B/?;;?@DCC@3;@>?;;;1,5>:5??9(+88<<?@4>3>><3<8>3>(+++3+:(44(

CCCFFFFFHHHHHJJJJJGIIJJIJJIIJIJJJJJJJIJJGIJJJJHJIIIJJGIIIIJIJHHHHFFFFFFEEDDCDDDDDDDDDDCDDCDDDCBDDDDD<

8?8?:0=A@;?DDEECAD94F;)?C8:D9D9;0B?@AD6@4--:?))..66(6((-(5,,,,(,>A(55<9+2>:(+(4>AD>(8((+4>(4>(:9:44+:

@@CFFFFFGHHHHIJJIJJIHIGIEGIGIIIIIHIJGGGIJAEHFFFFD>A6>CDDDDDDDCBBCDCDDA>?B4>?CDDCEC:CB38?CDCCACCCCCCC>

@@@FFFDFHHHFGIFJJJIGICHGGGGGJJIIIJJJ6@CGHGHHFCDFECEDECDDDDDDDDBDDDDDD<B<?>A<@BDDEEDDD?BDDDD@CACCCDDDE

CCCFFFFFHHGHHJJJJIJJJJJJGIGIJJJJJJJJIJJJJHHHFEFFEDEDECDDEDDDDDDDDDDDD<@?@>@?@BDCDEDCD??BCDDCDDDDCDEDC

CCCFFFFFHHHHHJIJJJJJJIJIJJIJJGIIJJJJJJJJJJJJJJJIJJJJJJJJIJIIIIJHHHHHDFFFFFA@CDFEEDDDDDCCDDD09CDDEDDDD

-----------------------------------------------------------------------------------

在1.2节中的每一个碱基都对应这一个质量值，质量值用ASCII码来表示。我们经过统计发现了三个特性用于压缩这些质量值的字符，分别是：

1. 这些字符总共只有40种，为ASCII码中从#到J连续的40个字符。
2. 这些字符的分布并不是均匀的，这些字符主要集中在对应的字符值最大的那部分，针对于部分数据的统计结果如图1所示。
3. 总是有很多连续重复出现的字符出现（如红色或蓝色的部分）。

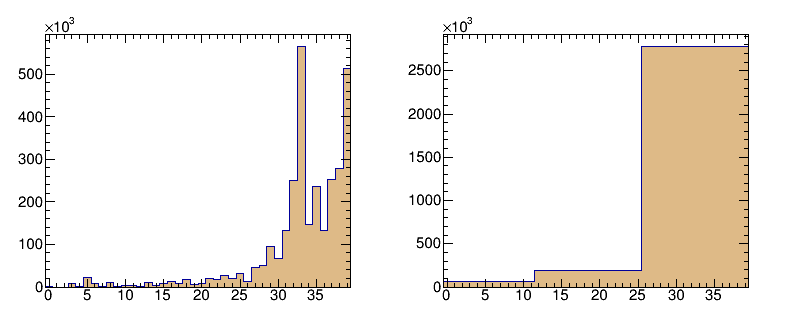


图 1 左图为这40个字符的分布图，可以明显看出，在字符对应的数值比较高的情况下，分布的较多。另一方面，更具体的针对于我们的压缩方案的分布显示在右图，我们可以明确地看出，多数字符分布在较大的三分之一处（ASCII码从‘=’到‘J’ 这部分）。

针对于这些特性，我们对质量值进行了压缩。首先我们面临的被压缩的数据是有40种可能的ASCII码，每一个ASCII码占用8个二进制位。而实际上只用到了40个字符，如果用二进制表示的话，需要6个二进制位（26=64 > 40 > 25 =32）。至少从这个角度说的话，可以至少把原来的质量值数据压缩到75%。

接下来我们针对于这些字符的分布规律来看（图1），很明显，这些字符大部分分布在26~39的范围内（这里面0对应于ASCII码中的‘#’，39对应于ASCII中的‘J’）。于是针对这种情况，我们把这些数据分为3类，如表1，

表 1 质量值ASCII码所对应的0~40的数字，以及变形后所对应的16进制数

|  |  |  |
| --- | --- | --- |
| ASCII码 | 对应的数值 | 变形后的16进制 |
| ‘#’ ~ ‘.’ | 0 ~ 11（共12个） | 1 ~ C |
| ‘/’ ~ ‘<’ | 12 ~ 25（共14个） | 1 ~ E |
| ‘>’ ~ ‘J’ | 26 ~ 39（共14个） | 1 ~ E |

我们认为用6位二进制码代表此40个字符仍是非常浪费的，根据图1中的分布，我们把这三类数分别在自己的类别内用16进制（16进制将占用4个二进制位）表示（为什么每一类我们只采用了最多14个，而不是排满了16个，后面会有所交代）。这样的话在一列质量值的情况下，我们只需要记录其中的‘>’~‘J’这些符号所代表的16进制数，用其中的1~E来表示。比如说，对于

------------------------原始数据ASCII码，每个字符需要8个二进制数来记录-----------------------

CC@FFFFFHHHHHJJJFHIIJJJJJJIHJIIJJJJJJJJIIGIJJIJJJIJJJIJIJJJJJJJJJJIJHHHHFFFDEEEEEEEEDDDCDDEEDDDDDDDDD

@@@FFFFFHHFFHGJJCGGHHIIGFHHGIGIF>GHIJJIJJFHGFCDD@CCDECDDEDD:<:<?CCABDAB8?::<@<>@ACCDD9?BDECCCDDDCDCCD

-----------------------------------------------------------------------------------

这样一个质量值序列（举例中的第一行和第三行），我们把其中的‘>’~‘J’转换为16进制数（1~E），而其他的字符分别用16进制中的0来表示(这些字符和16进制数标为红色为了明显一些)。即改写为，

-----------------------第一轮压缩的结果，每个字符都只需要4个二进制位来记录----------------------

774AAAAACCCCCEEEACDDEEEEEEDCEDDEEEEEEEEDDBDEEDEEEDEEEDEDEEEEEEEEEEDECCCCAAA89999999988878899888888888

444AAAAACCAACBEE7BBCCDDBACCBDBDA2BCDEEDEEACBA78847789788988000037756856030004024577880368977788878778

CECEACCEEB

-----------------------------------------------------------------------------------

这里这些被0代替的字符，将用十六进制表示并补充在队尾。比如说那些:<:<8::<9这些字符都分别被16进制0代替后，在队尾把这些字符仍然转化为16进制数CECEACCEEB，这些是根据表1的第二行来进行的（‘/’~‘<’）。如果在这些数码中仍然有包含于‘#’ ~ ‘.’的字符，那么我们就可以按照同样的替代原则，继续把他们用16进制0来代替，然后把真实结果表示在队列尾部。总而言之，16进制中的16不同的表示，我们把中间的14中用来记录所表示的字符，把0用来记录此处的字符其实是下一个16进制队列里的。这样看起来，表面上经过第一轮压缩之后，字符串的长度边长了。实际上，压缩之前的每一字符都至少需要6个二进制位来记录，而经过第一轮压缩之后，每一个字符都只需要4个二进制位来记录。而最初的原始数据，则是需要至少8个二进制位来记录。这样压缩之所以是可行的，主要是依赖于这些字符的分布是明显不均匀的。这样，通过第一轮压缩我们几乎可以把质量数据压缩至原来的50%。

-----------------------第二轮压缩的结果，每个字符都仍只需要4个二进制位来记录---------------------

774AF4CF4EEEACDDEF5DCEDDEF7DDBDEEDEEEDEEEDEDEF9DECF3AAA899F6888788998F8

444AF4CCAACBEE7BBCCDDBACCBDBDA2BCDEEDEEACBA788477897889880F337756856030004024577880368977788878778

CECEACCEEB

-----------------------------------------------------------------------------------

而同时有很多连续出现的相同的字符，比如AAAAA，比如我们就可以把它记为A4或者A5（A4表示A这样的数，将在后面补上4个同样的数码，也可以表示为A5，意思是A这样的数将重复出现5次），这样就大大减少了需要记录的内容，但是这样的情况下，我们将面临这样一个问题：我们该如何区分这里的4或者5到底是重复次数还是一个十六进制码。于是这里我们利用了前面提到的预留出来的最大的16进制码F。每当我们遇到F的时候，就表示，后面的数字并不是一个包含于队列的十六进制码，而是一个前面一位的重复此时，而F就是代表一个标签。经过这样的过程，可以把上面的数据继续压缩为上写的第二轮压缩结果那样。经过这样的压缩方式，可以把数据压缩到原始数据的40%左右。

综上所述，我们通过这三个小节描述了针对于三种不同的数据，我们分别根据数据的特点给出了一些针对性的压缩方法。总体来说可以压缩到原始数据的29%。至于之后，利用rar或者tar等成熟的压缩方式，可以进行进一步的压缩，最终可以压缩到原始数据的15%左右。利用已有压缩软件的压缩，我就不详细介绍了。

1. **使用方法（How to use）**

在这一章节，我将针对使用环境和一些对程序的粗糙的描述，来进行一下简单介绍。并直接给出如何使用这些代码。

**2.1 实现此算法的操作系统与程序语言**

1. **操作系统：**Linux（我实际使用的是windows10操作系统自带的Bash on Ubuntu on Windows，实际上任何Linux或Unix操作系统都是适用的，代码对操作系统几乎没有依赖。）
2. **程序语言：**C++。这里需要强调一下，我在程序中大量利用了char类型和int类型。其中char类型占用8个二进制位，而int类型占用4个char即32个二进制位。
3. **依赖的软件：**g++。我在程序中没有依赖任何额外的程序包，都是c++自带的程序库，此程序是一个clean的程序。

**2.2 程序介绍（c++）**

主要的算法实现都包含于source文件夹下，它们都有共同的名称空间namespace RonForALi，

Ron\_d19\_c10\_bin\_file.h

Ron\_d19\_c10\_bin\_file.cpp

Ron\_d19\_c20\_compress.h

Ron\_d19\_c20\_compress.cpp

在Ron\_d19\_c10\_file.h中我主要编写了12个class，名称和功能介绍如表2

表 2 Ron\_d19\_c10\_file.h中的各种类的功能

|  |  |
| --- | --- |
| BinData | 通用的用于记录二进制内容 |
| BinFile | 用于把BinData这种二进制写入文件和从文件中读取 |
| BasePair: public BinData | 衍生于BinData类，专用于碱基的压缩和解压 |
| WriteBasePair\_2\_BinFile | 把碱基压缩后的二进制信息BasePair写入文件类BinFile |
| ReadBasePair\_in\_BinFile | 把文件BinFile中的碱基压缩后的信息读入BasePair（解压） |
| Degree: public BinData | 衍生于BinData类，专用于质量值的压缩和解压 |
| WriteDegree\_2\_BinFile | 把质量值压缩的二进制信息Degree写入文件类BinFile |
| ReadDegree\_2\_BinFile | 把文件BinFile中的质量值压缩后的信息读入Degree（解压） |
| Info\_data | 用于存储一条描述信息的一个小部分 |
| Two\_Info\_data | 用于存储两条连续的描述信息的同样位置小部分，目的是比较 |
| All\_Info\_data | 压缩所有的描述信息，并同时存入二进制文件（一边压缩，一边写入文件） |
| Read\_All\_Info\_data | 把二进制文件中的描述信息读出，并逐条输出（按顺序逐条解压） |

上面列举的这些类，如果直接让用户去使用，是比较繁琐的，因此我写了两个函数把这些类直接整理起来。这两个函数分别用于压缩和解压，用户在使用的时候，只需要给出原始数据的文件名（路径名）和压缩后的数据所需要存储的目录名即可。这两个函数写于Ron\_d19\_c20\_compress.h。描述如下：

表 3用户用于解压和压缩的两个函数

|  |
| --- |
| void Compress\_2\_BinFile(const char \* fn, const char dirname);  这个函数用于把原始数据进行压缩，其中fn表示原始数据的名称。而被压缩的数据是以一个文件夹的形式存在，因此用户需要把可以用来存储压缩后的数据的文件夹名称告诉函数。但是需要注意的是，不能把两个原始文件压进同一个目录，那样做会导致第二个压缩的文件把第一个覆盖掉。这个压缩后的文件存储目录名称可以是不存在的，如果是不存在的，程序会自动创建一个。实际上，我们鼓励在一个已经存在的目录下给出一个新的名称来存储结果。 |
| void Uncompress\_4\_BinFile(const char \* dirname, const char \* fn);  这个函数用于把压缩到文件夹的数据解压到文件中，其中dirname表示的是需要被解压的文件夹（我们这里初步压缩就是以文件夹的形式存在），而fn是存储解压后文件名。如果fn这个文件已经存在，那么解压后的文件将覆盖之前的文件。如果这个fn不存在，程序将自动创建这个文件。 |

用户在使用这些方法的时候，需要把需要操作的内容写进一个main函数。这个事情我们举了一个例子，并把它写在了main.cpp中。当我们需要编译的时候，我并没有使用makefile的方式进行编译，而是用一个比较原始的方法。用户可以把main.cpp以及上面提到的4个文件名都包含在all.cpp中，这里我们写的例子就是：

//filename: main.cpp

#include <iostream>

#include <fstream>

#include "../source/Ron\_d10\_c10\_bin\_file.h"

#include "../source/Ron\_d10\_c20\_compress.h"

using namespace std;

int main()

{

RonForALi::Compress\_2\_BinFile("res\_b20\_0.txt", "dir");

RonForALi::Uncompress\_4\_BinFile("dir", "res\_b20\_uncompress.txt");

return 1;

}

//filename: all.cpp

#include "./source/Ron\_d10\_c10\_bin\_file.h"

#include "./source/Ron\_d10\_c10\_bin\_file.cpp"

#include "./source/Ron\_d10\_c20\_compress.h"

#include "./source/Ron\_d10\_c20\_compress.cpp"

#include "./source/main.cpp"

编译的时候，只需要在Linux操作系统的终端中的all.cpp所在的目录下输入

g++ -Wall -g all.cpp -lm -o a.out

在同一个目录下运行./a.out即可。

另一方面，如果我们需要做一个一劳永逸的事情，我们只需要bash shell脚本和一些简单的修改就可以了。这里我给出了一个简单而又粗糙的解决方案。我们首先需要删掉目录上可能存留的.out文件。之后运行

bash ./install.sh

在inshall.sh这个文件中，我们有如下内容

g++ -Wall -g all.cpp -lm -o a.out

g++ -Wall -g all\_4\_compress.cpp -lm -o compress.out

g++ -Wall -g all\_4\_uncompress.cpp -lm -o uncompress.out

这主要是用来编译三个文件，分别用于测试（a.out），压缩（compress.out）和解压（uncompress.out）。

如果我们直接运行./compress.out或者./uncompress.out的话，我们需要手动输入需要被压缩的文件名和保存压缩结果的目录名，这是有些繁琐的过程。我们把这些内容整合到了两个bash脚本中，分别为compress.sh和uncompress.sh。比如说我们有一个比较小的测试文档（实际上就是原始数据NA12878\_1.fastq的前30000个read单元），我们将其命名为res\_b20\_0.txt。如果我们想要把这个文件压缩到一个文件夹中res\_b20\_compressed，我们只需要进行如下操作

bash ./compress.sh res\_b20\_0.txt res\_b20\_compressed

这样就完成了这个压缩的过程。当需要解压的时候我们只需要

bash ./uncompress.sh res\_b20\_compressed res\_b20\_1.txt

这意味着我们把res\_b20\_compressed中的文件解压到res\_b20\_1.txt中。如果我们要比较压缩和解压是否相同，只需要运行

diff res\_b20\_0.txt res\_b20\_1.txt

在我测试的过程中，diff没有返回任何内容，意味这两个文件在压缩前和解压后是一样的，也就是说我们的压缩过程的确做到了无损压缩。

另一方面，如果用户想直接利用被压缩的数据，按照我们的算法是非常容易的，因为我们的算法是可以实现逐条解压的。但是目前来说，要求利用我们算法的程序语言也是c++，或者和c++有着良好的接口。这部分内容详细可以参考Ron\_d10\_c20\_compress.cpp中的RonForALi::Uncompress\_4\_BinFile函数。我在这里只是做一个简单的介绍。在RonForALi::Uncompress\_4\_BinFile函数中，在for循环中，每循环一次，代表读取一个read单元。这些read单元的结果分别以字符串的形式存储。

for(int i = 0; i < nlines; ++ i)

{

b1.Read();//描述信息存储于b1.fRes这个字符串中

out << b1.fRes << endl;

b2.Read();//碱基信息存储于字符串BasePair\_NS::BasePair\_2\_String\_res中

out << BasePair\_NS::BasePair\_2\_String\_res << endl;

out << "+" << endl;

b4.Read();//质量信息存储于字符串Degree\_NS::Degree\_2\_String\_Res中

out << Degree\_NS::Degree\_2\_String\_Res << endl;

}

并且值得一提的是，我们把各种信息都分别存储，从压缩文件中直接读取的时候，可以选择只读取碱基信息，或者描述信息，或者质量信息，并不要求用户每次都是读出所有的信息。这给用户使用这些信息带来了极大的方便。

1. **压缩效果（Results）**

**压缩时间**：我取了原始数据中的一部分作为估算压缩时间的，大小为76.2M，压缩到一个文件夹中的时间为3.55秒。原始数据为197G。那么估算得到，初步压缩的效率大致是21.5M/sec，对于一个197G的文件来说，初步压缩时间是153min。

**解压时间**：针对于上面的测试数据，解压时间大约是10.31sec。那么对于一个197G的大文件的压缩文件来说，解压时间大约需要444min。这里需要说明的是，如果需要利用这些数据，根本不需要进行解压，程序可以直接在被压缩的文件中按原来的顺序直接逐个给出原始数据。

**压缩率**：15%。初步压缩的效果大约是29%，再利用完善的软件再次压缩，压缩率可以达到15%。

1. **总结**

针对于三种类型的数据，根据其不同的特性，对其分别进行了压缩，各自压缩与独立的文件中。

初步压缩率达到了29%。再次利用软件可压缩至大约15%。

利用给出的c++函数，在不解压缩的情况下，可以独立地按顺序地读出三种数据信息中的任意一种。

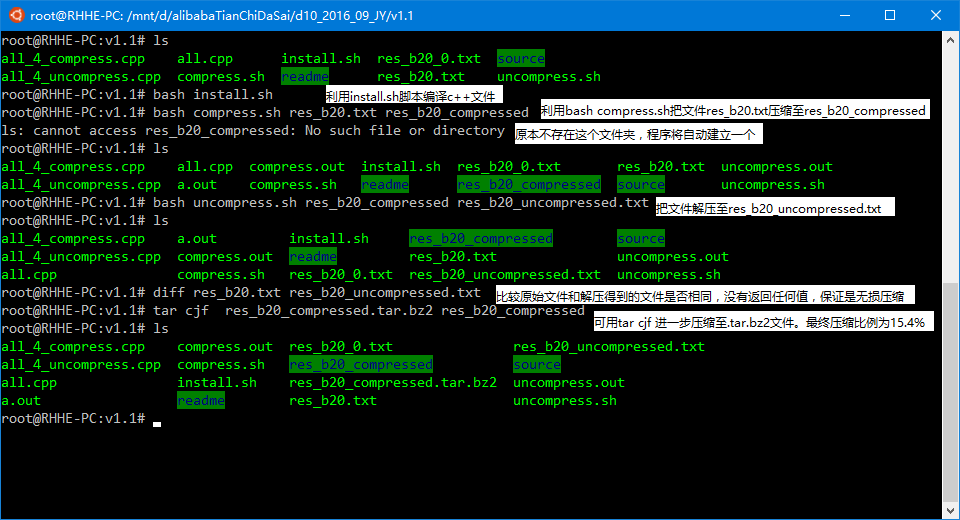
此算法仅仅使用了c++自带的函数，在linux操作系统上，除了g++编译器和系统自带的bash shell以外，并不需要其他任何额外的程序。

**致谢**

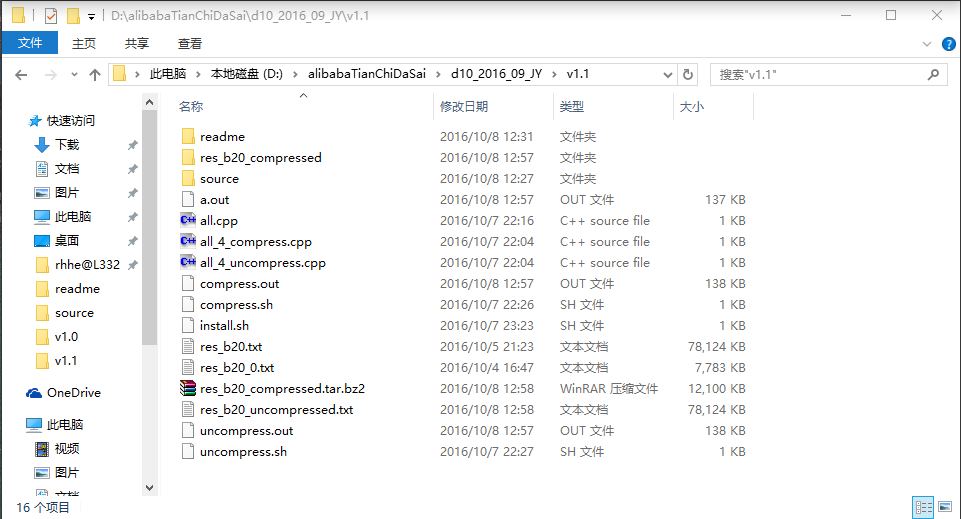
感谢Ph.D Hai-Ming Yuan的有价值的讨论与建议。

**附录**

附录1 测试流程



附录2 文件大小，res\_b20\_compressed.tar.bz2与res\_b29.txt，比例为15.4%



附录3 ASCII码表

