宏基因组组装项目报告

16307130194 陈中钰

1. 问题

1.1 问题描述：无参考、从头拼接基因组问题

- 输入 illumina 短序列、pacbio 长序列
- 输出拼接好的长片段

1.2 方法调查

- Greedy algorithm: 首先选择一个初始序列，通过启发式搜索方式寻找交叠部分最长的序列，并进行拼接，一直到不能再进行延长，输出拼接好的片段。这个方法依赖于硬匹配，复杂度高，运行时间长，而且结果不稳定，很难拼接出预期的结果。
- Overlap Layout Consensus: 以原始基因片段作为顶点，连接具有重叠长度超过阈值的点，然后在图中寻找 Hamilton 路径，连接该路径的基因片段，输出拼接好的结果。建图的时候需要两两比对，故复杂度为$O(n^2)$，而且构造 OLC 图的时候需要消耗大量内存。
- de Bruijn Graph: 把每个基因片段切割为 k 长的片段作为顶点，相邻两个片段之间有 k-1 长的重叠部分，并且连接相邻的片段，最后在 DBG 图中寻找欧拉路径，连接该路径的片段并输出拼接好的结果。

1.3 项目实现

由于以下两个原因，本次项目实现选了基于 DBG 图的方法拼接基因，

- 在 DBG 图中寻找欧拉路径可以简化为寻找最长路径，可以降低算法复杂度，而且还能保持不错的效果；
- 由于 DBG 图会把基因片段切割为 k 长的片段，所以可以同时利用短序列和长序列。

1.4 代码组织

本次项目是基于 python 实现的。

文件	内容
dbg.py	定义了节点 node 的类和 DBG 图的类
main.py	实例化 DBG 图，并循环求出最长路径，输出拼接好的 contig
utils.py	定义了数据读取的函数

1.5 运行方法

- 解压 data1.zip~data4.zip
- python main.py data1

（可以把 1 换成 1~4）

- 运行 data4 之前必须要运行 ulimit –s 8192000 把 stack size 调大

2. 方法

2.1 构建 DBG 图

2.1.1 伪代码：

```
build_DBG(G(V, E), data):
    for each read in data:
        rc_read <- get_reverse_complement(read)
        for i <- 0 to len(read) - k:
            add read[i: i + k], read[i + 1: i + 1 + k] into V
            add (read[i: i + k], read[i + 1: i + 1 + k]) into E
            add rc_read[i: i + k], rc_read[i + 1: i + 1 + k] into V
            add (rc_read[i: i + k], rc_read[i + 1: i + 1 + k]) into E
```

2.1.2 步骤描述：

- 对长度为 n 的 read（short1、short2、long 都使用）用固定长度为 k 的窗口滑动取出片段（kmer），窗口每次移动 1 个单位，则相邻两个 kmer 的重叠长度为 k-1，可以取出一共 n-k+1 个 kmer，并把全部 kmer 作为 DBG 图的顶点
- 连接相邻的两个 kmer，由前面 kmer 指向下一个 kmer
- 对 read 的**反向互补链**做同样的操作

2.1.3 具体实现

- 在把 read 的 k 长片段 kmer 作为结点加入图中时，同时需要建立 kmer->index 的 hash 表，在实际存储节点的时候只存 kmer 对应的 index，可以大大节省内存，而且寻找节点的时候可以通过 index 寻找；
- 在建立 kmer->index 的 hash 表的同时，还要建立一个 index->kmer 的数组，在后面需要从节点中还原出 kmer 用来拼接 contig
- DBG 图是由 index->node 对象的 hash 表实现的（不用数组的原因是后面有删除路径的步骤，因此需要支持删除的功能，所以使用 hash 表会更简洁），在最理想的情况下，可以通过$O(1)$的时间找到节点
- 构建 DBG 图的同时，需要记录每个节点使用的出现的次数，在后续会用到

2.2 寻找最长路径

2.2.1 伪代码

```
get_longest_path(G(V, E)):
    max_depth <- 0
    max_v <- None
    for v in V:
        depth <- get_depth(v)
        if depth > max_depth:
            update max_depth, max_child
    path <- []
    while max_v is not None:
        append max_v into path
        max_v <- max_v.max_child
    return path
```

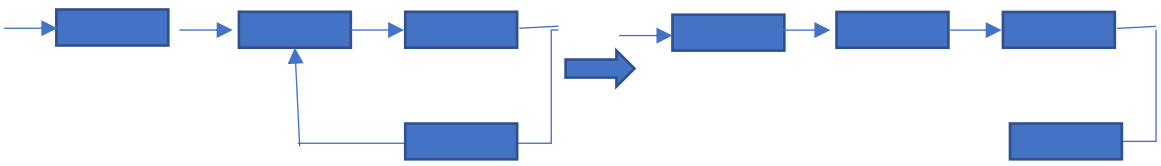
```
get_depth(v):
    if v.visited is False:
        v.visited <- True
        max_depth <- 0
        max_child <- None
        for child in v.children:
            depth <- get_depth(child)
            if depth > max_depth:
                update max_depth, max_child
        v.depth <- max_depth + 1
        v.max_child <- max_child
    return v.depth
```

2.2.2 步骤描述

- 通过动态规划 get_depth()递归函数，计算出每个节点 v 在 DBG 图中最远可以到达的距离 max_depth，并记录在节点信息中，同时记录走向哪一个子节点才能达到该最远距离，可以在$O(V)$完成；
- get_longest_path()遍历每一个节点，找出最远可以到达的距离最大的点，可以在$O(V)$完成；
- 以该点作为起点，顺着可以到达最远距离的子节点信息 max_child，还原出 DBG 图的最长路径，可以在$O(V+E)$完成；
- 而路径中前一个 kmer 的后 k-1 部分和后一个 kmer 的前 k-1 部分是重叠的，把路径拼接为 contig，可以在$O(path)$完成并输出

2.2.3 具体实现

- 遍历节点 v 的子节点的顺序在**讨论部分**进行叙述；
- 每个节点的 max_depth、max_child 都初始化为 0、None，在每次求最长路径的之前，都要像这样重置
- 动态规划递归计算每个点最远可达距离时，递归的不是整个节点，只需要递归节点的 index，然后通过 hash 表可以获得对应的节点
- 该算法由于经过了的点就不再重复求解最远可达距离，因此相当于 DBG 图中的环被断开了



2.3 删除上述路径

2.3.1 伪代码

```
delete_path(G(V, E), path):
    for v in path:
        delete v in V
    for v in V:
        for child in v.children:
            if child in path:
                delete child in v.children
```

```
delete_path(G(V, E), path):
    for v in path:
        for father in v.father:
            delete v in father.children
        for child in v.children:
            delete v in child.father
        delete v in V
```

2.3.2 步骤描述

方法一

- 把 path 中的节点在图中删掉
- 如果图中剩下的点有 child 是 path 中的点，把该 child 去掉

方法二：

- 遍历 path 中的点，把它和子节点、父节点的连接去掉，再去掉这个点
- 这个方法的复杂度更低，但是需要在构建 DBG 图的时候记录父亲节点的信息。

2.3.3 具体实现

- 遍历的是 index，通过 index 获得节点

2.4 继续寻找下一条最长路径，并输出拼接好的 contig

- 寻找下一条最长路径前，需要重置每个节点的 max_depth、max_child 信息
- 由于 NGA 要求 contig 和真实结果重叠的部分要超过比例阈值，这样的 contig 才是有效的，而不断寻找最长路径并删除，会使最长路径越来越短，通过实验证明，**只需要输出前 20 条最长路径对应的基因即可，接下来输出的基因都太短而对结果没有影响**。

3. 效果

以下分别为 data1~data4 的最终结果：

排名	昵称	提交时间	提交次数	Genome_Fraction(%)	Duplication ratio	NGA50	Misassemblies	Mismatches per 100kbp
20	crayon	2019/06/18 11:27:47pm	22	99.886	1.982	9118.8	2.0	0
18	crayon	2019/06/18 11:29:13pm	7	99.942	2.0052	9129.2	3.0	0
20	crayon	2019/06/18 11:29:59pm	9	78.6	1.6	7859.2	0.0	0
5	crayon	2019/06/18 11:30:42pm	10	78.2948	1.607	55757.8	11.0	0

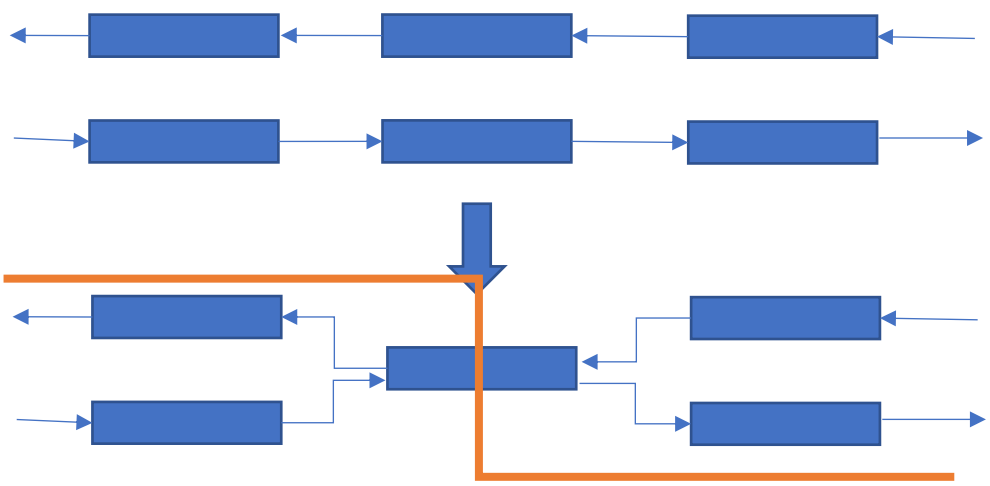
4. 讨论

4.1 为什么要添加反向互补链？

- 因为 DNA 双链的两条链的复制总是从 5'开始的，如果把反相互补链也用于构造 DBG 图，可以弥补一些单链的空缺，可以连接出更长的 contig，结果会更好；
- 但是缺点是，duplicatation ratio 会保持在 2 左右，因为把所有的 read 的反相互补链都用上了

4.2 为什么 k 要选择奇数？

- 如果 k 选择偶数，则 kmer 的反向互补会与自身相同（奇数的时候不会），会出现原链和反向互补链混淆的问题。如果在找路径的时候，选择了橙色的路径，那么就有混淆的问题了。



4.3 k 的选择？

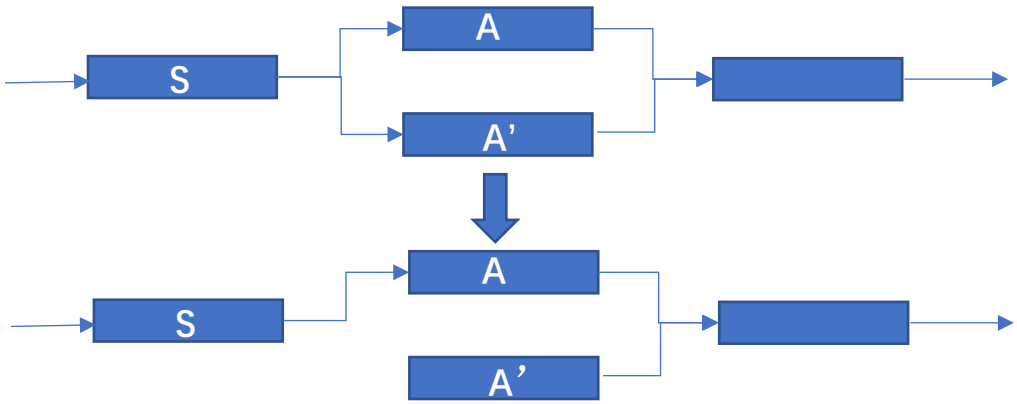
- 尝试了 21、23、25、27、29，最后发现，在四个数据集中，都是在 k=25 的时候最好，我猜测这个值和 read 的获得方式有关。、
- 上述结果都是在 k=25 的时候的最好结果

4.4 处理 tip 的错误结构

- tip 的长度是短的，比正链要短，由于算法只输出了最长的若干条 contig，因此最终输出并不包含 tip，所以不需要处理

4.5 处理 bubble 的错误结构

- 如果在序列中有 1 个字符错了，则会形成这样的 bubble，A 和 A'中每个对应的 kmer 都只有 1 个对应字符是错的，且 A 和 A'长度一样。设 A 是正确的，而 A'是错的。



- 在递归计算最远可达距离时，**要从子节点的最远可达距离中取最大的值，而遍历子节点的时候，要按照节点出现次数从大到小的顺序遍历**，因此，如果遇到了其他相同大小的最远距离子节点，但是出现次数更少，这时候就不会再更新最远可达距离和对应的路径方向。
- 由于 A 和 A'的长度一样，**我认为 A 的第一个 kmer 的出现次数应该比 A'的第一个 kmer（错了一个字符）的出现次数要多**，因此 S 的最后一个 kmer 遍历子节点的时候先遍历 A 的第一个 kmer，就更新了最远可达距离和走向是向 A 的第一个 kmer，接着遍历 A'的第一个 kmer 时，虽然最远可达距离一样，但是出现次数更少，不再更近最远距离。寻找最长路径的时候，就会选择 S->A 的路径，于是解决了 bubble 的错误问题。

5. 结论

5.1 DBG 图的优点

- 计算复杂度低，前 3 个数据都能在 5min 内获得前 20 条最长路径，而 data4 稍长，需要 10min 左右
- DBG 图构建方式简单，找最长路径思路直观，直观上感觉很符合基因拼接的规律
- 拼接效果不错，还能解决部分基因的错误

5.2 DBG 图的缺点

- 由于找最长路径的时候需要用动态规划递归求解每个点的最远可达距离，函数递归的深度很大，为了能运行 DBG 图，需要设置更大的 recursion limit，而且当数据量很大的时候（data4），还需要设置更大的 stack size 才能正常运行。（可以通过消除尾递归来减轻递归压力）

```
import sys
sys.setrecursionlimit(100000)
```

```
ulimit -s 8192000
```