**哈尔滨工业大学计算学部**

**课程实验报告**

课程名称：生物信息学

课程类型：选修

项目名称：基因组片段拼接算法

项目名称：

班级：2103601

学号：2021112845

姓名：张智雄

|  |  |  |
| --- | --- | --- |
| **设计成绩** | **报告成绩** | **指导老师** |
|  |  | **刘博** |

1. **实验目的**

给定基因组序列（大肠杆菌E. Coli, 长度约5Mbp）作为输入，建立算法和系统，可以完成对基因组测序片段的拼接，形成contig一系列序列。

* 采用基于de Bruijn graph和欧拉路径的拼接算法
* 根据输入数据，自行构建de Bruijn graph，并在其中找到所有的unipath作为contigs输出。

1. **实验原理**

测序产生的小片段需要拼接成更长的大片段（contig），进而组装成完整的基因组。这个拼接的过程基于片段之间的重叠关系。在同一基因组中，由于随机切割产生的小片段很可能存在重叠部分，如下图所示：

图片包含 图形用户界面

描述已自动生成

利用这些重叠的片段，我们可以找到正确的路径，将这些测序产生的小片段连接起来。然而，测序可能存在误差。例如，假设有两个长度约为100bp的序列，它们之间有约30bp的重叠，那么要求这两个片段的重叠部分都没有错误才能将它们接合起来。然而，由于测序误差的存在，保证这种准确性非常困难。

为了解决这个问题，可以采用动态规划的方法，但动态规划的时间复杂度较高。因此可以将片段划分为-mer片段，然后利用deBruijn图找出-mer之间的联系。在deBruijn图中，每个-mer被视为一个节点，该-mer的前-mer作为入边，后-mer作为出边。如果一个-mer的后-mer与另一个kmer的前-mer相同，则这两个节点可以通过一条边相连。

**2.1 Kmer Counting**

在使用deBruin graph 算法之前，我们需要进行-mer counting步骤，可以使用Jellyfish或KMC等工具来进行快速的-mer counting。本实验直接使用KMC在条件下生成的-mer counting数据。

为了便于之后进行统计以及有向边的寻找和压缩内存，本实验基于实验一使用两个比特位(bit)来存储碱基[腺嘌呤(A)、胞嘧啶(C)、鸟嘌呤(G)和胸腺嘧啶(T)]，于是每一条-mer(64 bits)恰好可以存储在一个无符号长整型当中，便于之后进行碱基转化以及有向边的统计。

此过程由uint64\_t C2B(std::string seq)函数完成，其具体功能即将长度为32的字符串表示的DNA序列转换为一个64位的整数表示，其中每个字符（A、C、G、T）都用二进制编码表示，各碱基具体编码如下：

表格 1 碱基编码

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 字符 | A | C | G | T | 其他 |
| 对应编码 | 00 | 01 | 10 | 11 | 非法字符 |

2.2 节点以及边的构建

**2.2.1节点的构建：**

在构建DeBruijn图的过程中，对于-mer及其邻居的查找、插入等操作对拼接算法的效率至关重要。由于std::unordered\_map能够提供较快的平均查找、插入和删除性能，因此本实验将每个节点存储在一个unordered\_map中，这个容器的底层实现是哈希表，能够在的时间内完成增添、删除和查找操作。

同时，在这个场景中，我们不需要按键值顺序存储数据，因此使用std::unordered\_map不会带来额外的不便。

**2.2.2边的构建：**

在deBruijn图中，我们定义一个节点和另一个节点之间存在着一条有向边，如果当前一个节点的后个序列与后一个节点的前个序列相同。因为我们将每个-mer存储在一个无符号长整型中，所以可以通过位运算在的时间内完成是否存在有向边的判断。具体获得前后缀的计算如下：

get prefix：kmer >> 2； get suffix：kmer<< 2 >> 2；

根据序列的特性，我们可以通过将一个节点的长的后缀与分别进行结合，然后判断是否在哈希表中，来判断这个节点是否存在一条有向边。通过这种方法，我们将构建有向边的时间复杂度从转换成了，大大节省了算法的时间开销。

文本

描述已自动生成

图 1 边构建

2.3 欧拉路径的寻找

Hierholzer算法是一种高效的寻找欧拉路径或欧拉回路的算法。**欧拉路径指的是在图中通过每条边恰好一次且仅一次的路径，而欧拉回路是一种特殊的欧拉路径，它的起点和终点相同。**该算法由Carl Hierholzer于1873年提出。

其基本思路是从图中的一个顶点出发，沿着未访问过的边前进，直到到达一个无法继续前进的顶点。在这个过程中经过的边和顶点形成了一个回路。然后，从回路中找到一个仍有未访问过的边的顶点，从该顶点出发继续前述过程，直到所有的边都被访问过。最后，将这些回路合并即可得到一个欧拉回路或欧拉路径。具体地，Hierholzer算法的详细步骤如下：

1. **选择一个起始顶点**。对于欧拉路径问题，选择一个具有奇数度的顶点（如果存在）作为起始顶点；对于欧拉回路问题，可以选择任意顶点作为起始顶点。
2. 从当前顶点出发，**沿着未访问过的边前进**，直到无法继续前进。这个过程中经过的边和顶点形成了一个回路。在访问边的过程中，将边标记为已访问。
3. 在当前回路上找到一个仍有未访问过的边的顶点，将其作为**新的起始顶点**。如果没有这样的顶点，跳到步骤5。
4. 从**新的起始顶点出发，重复步骤2和步骤3**，直到所有的边都被访问过。然后，将新形成的回路与之前的回路合并。合并方法是将新回路插入到原回路中新起始顶点的位置。
5. 所**得到的回路**就是欧拉路径或欧拉回路。

文本

描述已自动生成

图 2 Hierholzer算法

1. **测试结果及分析**

首先查看-mer counting（）之后生成的txt文件如下：

文本

描述已自动生成

图 3 -mer counting（）

而后进行编译，成功读取文件并开始建立deBruijn图

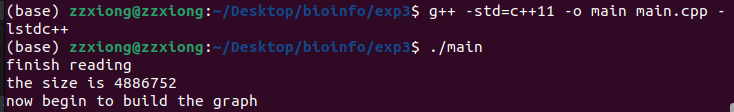


图 3 编译过程

由于原来的序列当中存在着gap，所以不能直接通过vim的搜索功能进行验证，只能通过目测法进行正确性的验证。而后生成的contigs如下所示：

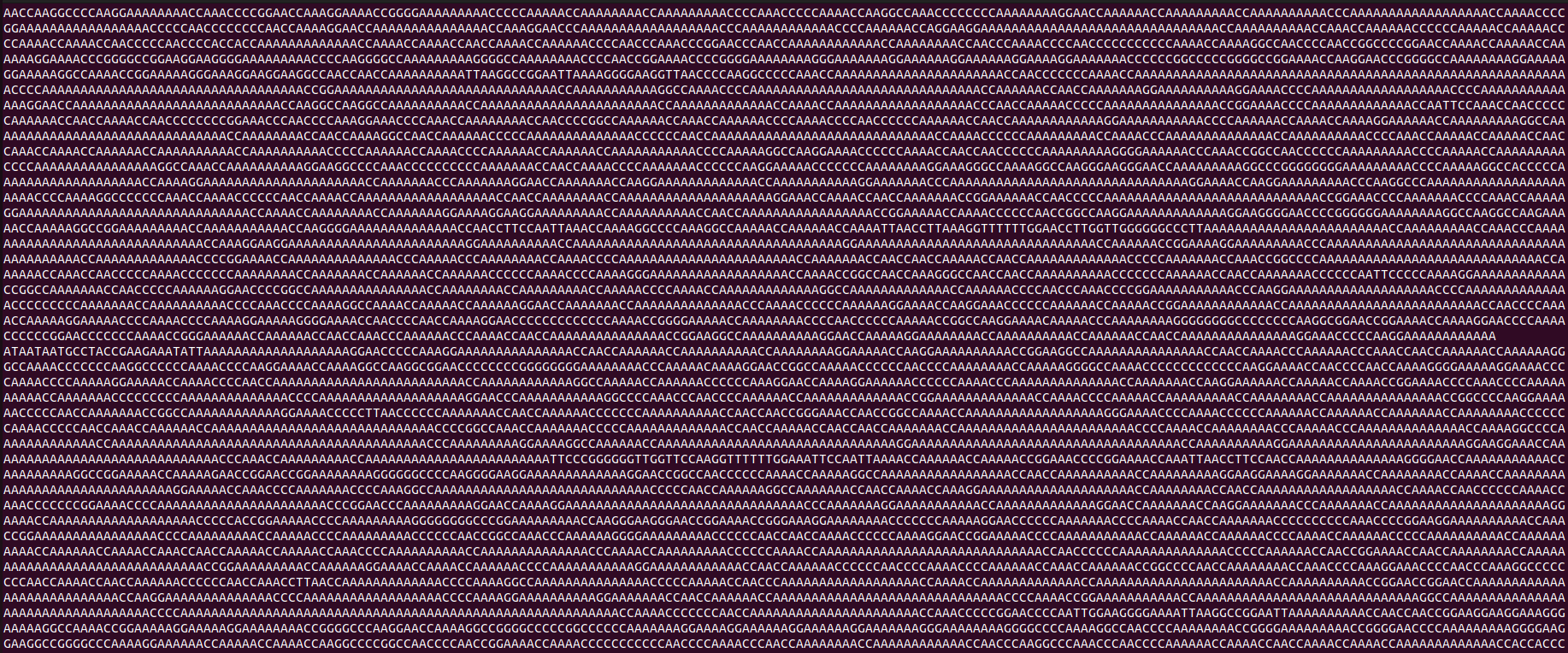


图 4 实验结果

1. **经验体会**

在这个实验中，我们深入探索了De Bruijn图和Hierholzer算法在基因组拼接中的应用。通过实践，我们获得了以下主要收获：

* 基因序列的优化表示：学会了将基因序列从字符串表示转换为二进制表示，以优化空间和计算时间。
* De Bruijn图的构建：了解了如何构建De Bruijn图，并学会了在C++中使用unordered\_map高效地表示这种图结构。
* Hierholzer算法的应用：掌握了Hierholzer算法，并理解了它如何应用于寻找欧拉路径。此外，还学会了如何将这种算法应用于基因组拼接问题。
* 基因拼接序列的还原：学会了如何从欧拉路径中还原出基因拼接序列。
* 实践中的算法实现：了解了如何在C++中实现整个实验流程，包括读取输入文件、构建De Bruijn图、找到欧拉路径以及输出基因拼接序列。

总的来说，这次实验为我们提供了丰富的生物信息学和算法实践经验。它不仅提升了我们的编程技能，还加深了我们对生物信息学领域问题和解决方案的理解。通过这次实验，我们认识到了理论知识与实际问题之间的联系，以及如何将这些理论知识应用于实际生物学研究中。

1. **附录：源代码（带注释）**

**main.cpp**

|  |
| --- |
| #include <iostream>  #include <string>  #include <vector>  #include <algorithm>  #include <fstream>  #include <omp.h>  #include <bitset>  #include <unordered\_map>  #include <sstream>  const int N = 11260625;  std::unordered\_map<uint64\_t, int> kmer\_counts; // kmer, count  //graph  std::unordered\_map<uint64\_t, std::vector<uint64\_t>> graph; // kmer, neighbors  void printcode(uint64\_t seq){  std::bitset<64> x(seq);  std::cout << x << std::endl;  }//print binary code, for debug  uint64\_t C2B(std::string seq){//convert char to binary,to save space and compute time  uint64\_t Bseq = 0;  uint64\_t cur\_c = 0;  for(int i = 0; i < 32; i++) {  if(seq[i] == 'A'){  cur\_c = 0;  }  else if(seq[i] == 'C'){  cur\_c = 1;  }  else if(seq[i] == 'G'){  cur\_c = 2;  }  else if(seq[i] == 'T'){  cur\_c = 3;  }  Bseq = Bseq | cur\_c;  if(i != 31) Bseq = Bseq << 2;  }  return Bseq;  }  uint64\_t prefix(uint64\_t kmer)//get prefix  {  return kmer >> 2;  }  uint64\_t suffix(uint64\_t kmer)//get suffix  {  return kmer<< 2 >> 2;  }  std::string get\_string(uint64\_t kmer){//get string from binary  std::string seq;  for(int i = 0; i < 32; i++){  uint64\_t cur\_c = kmer >> (62 - 2\*i);  cur\_c = cur\_c & 3;  if(cur\_c == 0){  seq += 'A';  }  else if(cur\_c == 1){  seq += 'C';  }  else if(cur\_c == 2){  seq += 'G';  }  else if(cur\_c == 3){  seq += 'T';  }  }  return seq;  }  char last\_ch(uint64\_t kmer){//get last char  uint64\_t cur\_c = kmer & 3;  if(cur\_c == 0){  return 'A';  }  else if(cur\_c == 1){  return 'C';  }  else if(cur\_c == 2){  return 'G';  }  else if(cur\_c == 3){  return 'T';  }  return 'N';  }  std::vector<uint64\_t> get\_neighbors(uint64\_t kmer) {  //get neighbors  std::vector<uint64\_t> neighbors;  uint64\_t suffix\_kmer = suffix(kmer);  for (int i = 0; i < 4; ++i) {  uint64\_t neighbor = (suffix\_kmer << 2) | i;  if (kmer\_counts.count(neighbor)) {  neighbors.push\_back(neighbor);  }  }  return neighbors;  }  void build\_de\_bruijn\_graph() {  for (const auto& kmer\_count : kmer\_counts) {  //kmer\_count is a pair, first is kmer, second is count,this for loop is to build graph  uint64\_t kmer = kmer\_count.first;  std::vector<uint64\_t> neighbors = get\_neighbors(kmer);  if (!neighbors.empty()) {  graph[kmer] = neighbors;  }  }  }  void hierholzer(uint64\_t current\_kmer, std::unordered\_map<uint64\_t, std::vector<uint64\_t>>& graph, std::vector<uint64\_t>& path) {  //hierholzer algorithm  //this function is to find a eulerian path  //how can find: https://www.geeksforgeeks.org/hierholzers-algorithm-directed-graph/  //thanks a lot to the author  while (!graph[current\_kmer].empty()) {  uint64\_t next\_kmer = graph[current\_kmer].back();  graph[current\_kmer].pop\_back();  hierholzer(next\_kmer, graph, path);  }  path.push\_back(current\_kmer);  }  std::vector<uint64\_t> find\_start\_kmer(const std::unordered\_map<uint64\_t, std::vector<uint64\_t>>& graph) {  //find start kmer, which has out degree but no in degree  std::vector<uint64\_t> start\_kmers;  for (const auto& entry : graph) {  if (entry.second.size() > 0) {  start\_kmers.push\_back(entry.first);  }  }  return start\_kmers;  }  std::string path\_to\_contig(const std::vector<uint64\_t>& path);  std::vector<std::vector<uint64\_t>> eulerian\_paths(std::unordered\_map<uint64\_t, std::vector<uint64\_t>>& graph) {  //find eulerian path  std::vector<std::vector<uint64\_t>> paths;  std::vector<uint64\_t> path;  std::vector<uint64\_t> start\_kmers = find\_start\_kmer(graph);  for (const uint64\_t& start\_kmer : start\_kmers) {  hierholzer(start\_kmer, graph, path);  std::reverse(path.begin(), path.end());  if(path.size() > 1000){  std::cout << path\_to\_contig(path) << std::endl;  }  }  return paths;  }  std::string path\_to\_contig(const std::vector<uint64\_t>& path) {  std::string contig = get\_string(path.front()).substr(0, 31);  for (const uint64\_t kmer : path) {  contig += last\_ch(kmer);  }  return contig;  }  int main() {  std::ifstream fin("./NC\_008253.se.5end.forward.27mers.txt");  std::string line;  int cnt\_tmp = 0;  int i = 0;  // delim within '\t'  while (std::getline(fin, line)) {  std::istringstream iss(line);  std::string part1, part2;  std::getline(iss, part1, '\t');  std::getline(iss, part2);  cnt\_tmp = std::stoi(part2);  uint64\_t kmer = C2B(part1);  kmer\_counts[kmer] = cnt\_tmp;  i++;  }  std::cout << "finish reading" << std::endl;  std::cout << "the size is " << i << std::endl;  std::cout << "now begin to build the graph" << std::endl;  build\_de\_bruijn\_graph();  // C2B("GCCTGGGCGAATTATTTCTCCGAACACACAAC");  std::cout << "finish building the graph" << std::endl;  std::cout << "now begin to find the eulerian path" << std::endl;  std::vector<std::vector<uint64\_t>> paths = eulerian\_paths(graph);  std::cout << "finish finding the eulerian path" << std::endl;  //print the contigs  for(const auto& path : paths){  std::cout << path\_to\_contig(path) << std::endl;  }  return 0;  } |