

**毕 业 论 文(设 计)**

**论文（设计）题目:**

**基于流分解的转录组组装算法实现**

姓 名 **张建喜**

学 号 **201400220057**

学 院 **山东大学软件学院**

专 业 **软件工程**

年 级 **2014级**

指导教师 **冯好娣**

2014年 5月 11日

**目录**

[摘要 1](#_Toc514612611)

[ABSTRACT 2](#_Toc514612612)

[第1章 绪论 3](#_Toc514612613)

[1.1 研究背景 3](#_Toc514612614)

[1.2 研究意义 3](#_Toc514612615)

[第2章 转录组组装总体流程设计 7](#_Toc514612616)

[2.1转录组组装两大模块 7](#_Toc514612617)

[2.1.1 构建splicing graph 7](#_Toc514612618)

[2.1.2 转录本的鉴定与定量 8](#_Toc514612619)

[第3章 转录组组装详细设计 10](#_Toc514612620)

[3.1 数据结构设计 10](#_Toc514612621)

[3.2 图结构设计 10](#_Toc514612622)

[3.2.1 addChildren(int child) 11](#_Toc514612623)

[3.2.2 getChildren() 12](#_Toc514612624)

[3.2.3 isChild(int child) 12](#_Toc514612625)

[3.2.4 deleteChild(int child) 12](#_Toc514612626)

[3.2.5 clearChildren() 13](#_Toc514612627)

[3.2.6 addParents(int parent) 13](#_Toc514612628)

[3.2.7 getParents() 13](#_Toc514612629)

[3.2.8 isParent(int parent) 14](#_Toc514612630)

[3.2.9 deleteParent(int parent) 14](#_Toc514612631)

[3.2.10 clearParents() 15](#_Toc514612632)

[3.2.11 setSequence(String seq) 15](#_Toc514612633)

[3.2.12 getSequence() 15](#_Toc514612634)

[3.3 基本操作设计 15](#_Toc514612635)

[3.3.1 intToBase(int c) 15](#_Toc514612636)

[3.3.2 baseToInt(char base\_val) 16](#_Toc514612637)

[3.3.3 intvalToKmer(long int\_val,int kmer\_length) 16](#_Toc514612638)

[3.3.4 kmerToIntval(String kmer) 16](#_Toc514612639)

[3.3.5 get\_forward\_candidates(long seed\_kmer, Map<Long, Vector<Integer>> kmer\_hash) 17](#_Toc514612640)

[3.3.6 get\_reverse\_candidates(long seed\_kmer, Map<Long, Vector<Integer>> kmer\_hash) 18](#_Toc514612641)

[3.4 读取输入数据，构建HashMap 19](#_Toc514612642)

[3.4.1 提取输入文件中的reads信息 19](#_Toc514612643)

[3.4.2 将提取的reads信息构建成HashMap 19](#_Toc514612644)

[3.5 根据kmer\_hash构建splicing graph 20](#_Toc514612645)

[3.5.1 对kmer\_hash中的k-mer进行排序 20](#_Toc514612646)

[3.5.2 构建树干 20](#_Toc514612647)

[3.5.3 正向构图 22](#_Toc514612648)

[3.5.4 反向构图 27](#_Toc514612649)

[3.5.5 重置父节点 30](#_Toc514612650)

[3.6 根据splicing graph运行最大流算法 30](#_Toc514612651)

[3.6.1 计算每个顶点的权重 30](#_Toc514612652)

[3.6.2 对路径权重进行排序 31](#_Toc514612653)

[3.6.3 计算流网络中边的权重 32](#_Toc514612654)

[3.6.4 重构流网络运行最大流算法 36](#_Toc514612655)

[第4章 改进与总结 39](#_Toc514612656)

[4.1 结果分析 39](#_Toc514612657)

[4.2 改进 42](#_Toc514612658)

[4.3 总结 44](#_Toc514612659)

**基于流分解的转录组组装****算法实现**

# 摘要

目前我们使用的对转录组进行测序的方法通常会产生超过2亿个短序列。我们实现了一个基于流网络的计算方法，其利用最优化理论和从头组装（可选）将这些复杂的数据集合组装成转录本。不同于其它转录组组装方法，该方法将重建转录本和转录本的定量分析同时进行，运行时间少，空间复杂度小。其具体的实现方法是，通过输入的reads信息构建k-mer字典，再利用k-mer字典构建splicing graph，splicing graph上的每一条路径都代表了一条可能的转录本。但是为了提高该算法的精度，我们需要在该路径基础上创建流网络并执行最大流算法来对转录本进行定量分析，以此得到转录本的转录丰度，确定哪些转录本是我们最终预测的转录本。在模拟数据上运行该算法，分析该算法的运行结果可知，该算法可以正确组装出10%-30%的转录本。

**关键字**：转录组组装、定量分析、精度、灵敏度

# ABSTRACT

Today’s sequencing technology often produce more than 200 million short sequences and these short sequences are called reads. We implemented a computational method that applies a flow network algorithm to assemble these complex data sets to transcripts.This algorithm uses optimization theory and optional de novo assembly. Unlike other transcriptome assembly methods, this method assembles transcripts and estimates their expression levels simultaneously. However, this method has high accuracy and sensitivity.

The specific implementation of this method is to construct a splicing graph by the information of reads sets. Paths through the splicing graph represent possible splice variants or possible transcripts. For each path we create a separate flow network to estimate its expression level using a maximum flow algorithm. The algorithm implemented in this article can correctly assemble 10%~30% of true transcripts.

**keyword**：transcriptome assembly、quantitative analysis、accuracy and sensitivity

# 第1章 绪论

## 1.1 研究背景

过去的众多研究已经表明，真核生物的转录组的多样性程度日益加剧。大多数的转录元件由无数的ncRNAs（non-protein-coding RNA genes 非编码蛋白质RNA）编码而成，并且这些有复杂的表达和调节模式的转录元件并不会被翻译为蛋白质。进一步来讲，多外显子-蛋白质编码基因的转录本的90%以及ncRNAs的30%，都经历了选择性剪切。对于人类基因组而言，想要对人类基因组中所有的转录本拥有一个全面的认识，我们还需要经历一个漫长的研究过程。对于人类物种尚且如此，对于非人类物种，研究前景就更加迷茫了。

## 1.2 研究意义

转录组是特定的某个物种或者特定的某个细胞类型产生的所有转录物的集合，如果我们可以知道转录组的序列，那么会对研究基因功能、基因结构、甚至是研究各种疾病等方面提供强有力的支持。但是如何对转录组进行测序是一直以来困扰我们的难题。近几年，随着RNA-seq的出现，基因探测方法也被革新。RNA-seq是一种对转录基因序列进行测序的高通量测序方法，其基于二代测序技术进行cDNA测序，全面快速地获取某一物种特定器官或组织在某一状态下的几乎所有转录本。从前我们通过识别剪切位点、编码区域和其他信号来从头对基因建模。但是现在，这些方法已经被使用RNA-seq数据作为输入进行建模这样一个更加高效的方法所取代。每进行一次RNA-seq，就会产生超过2亿个短序列，我们称这些短序列为read，每个read大约为100-150bp（碱基）的长度。利用这些reads进行转录组的组装和定量，主要需进行三个步骤：reads映射、转录组重建和表达水平量化。由于不同的转录本有不同的序列覆盖度，并且来自于同一个基因座的不同转录本可以共享外显子，以上两个原因使得我们很难准确地组装多种亚型。就算我们已经知道了转录本的具体结构，为这些转录本进行定量分析也是一个不小的挑战。转录本之间共享外显子、密切的旁系同源关系导致的read映射不明确以及基因的低水平表达都是难以准确定量的影响因素。因此将这些庞大的reads数据组装成完整的转录本是一项非常艰难的任务。

尽管目前已经有了很多解决转录组组装问题的方法、解决表达定量的方法或者解决以上两个问题的方法，但是为了生成一致的、高精度的解决方案我们还需要做出不懈的努力。此外，多种亚型和新型剪切变体的表达极大地影响了转录组重建的准确性。因此，我们需要一个更加高效、准确的方法来重建转录组。

目前的转录组组装有两种方法，一种是基于参考基因的组装，一种是无需参考基因的从头组装。无论是哪种组装方式，我们都要利用reads短序列构建图模型以表示每个基因的所有可能的亚型，这样的图模型叫做splicing graph(剪切图)，剪切图中的顶点代表了完整外显子或者部分外显子，图中的路径代表了可能的剪切变体，本次设计采用的便是splicing graph。目前被广泛使用的转录组组装软件Cufflinks使用的是overlap graph(重叠图)，在重叠图中，测序片段（sequenced fragments）作为节点，如果两个节点有重叠并且具有兼容的拼接模式，则连接这两个点。Cufflinks和其他的转录组组装软件采用不同的标准去解析这些图模型，以Cufflinks为例，Cufflinks采用基于简约的算法，从图中生成能够覆盖所有reads的最少数量的转录本。当我们使用简约原则从图中生成最少数量的转录本时，此时往往会忽略与底层转录本相关的覆盖水平，得到的转录本往往也会是错误的、不完全的。因此简约原则虽然好，但是它并没有考虑到转录本的转录丰度，并且可能不会重建出正确的亚型组。显然，如果生成了错误的转录本，那么根据错误的转录本计算出的表达水平也不会是正确的。

Cufflinks将寻找最少数量的转录本和估计这些转录本的表达水平分开进行，基于流分解的转录组组装算法则同时进行转录本的组装和转录本表达水平的计算，我们定义该算法为StringTie。首先我们利用reads信息创建splicing graph，splicing graph 上的每一条路径都代表了一个可能的转录本，再对每一个转录本创建一个单独的流网络并采用最大流算法估计该转录本的表达水平。虽然流网络在其它转录组组装软件中被使用过，如Traph，但是与StringTie不同的是Traph解决的是最小流问题。

尽管StringTie可以使用已知的基因坐标来指导转录组的组装过程，但是我们把讨论的重点放在如何准确地组装和量化转录本，无论先前的注释是否已知。我们在没有注释的情况下评估StringTie、Cufflinks、IsoLasso、Scripture和Traph，以上软件再加上MITIE是唯一可以在没有注释的情况下进行转录组组装的软件。我们也试图对MITIE进行评估，但是过长的运行时间使其被排除在评估的行列之外。实际上，经过了一个月的计算，MITIE并没有对研究中使用的任何数据集产生结果。由于我们不可能确切地知道哪些转录本在现实中是实际存在的，因此在评估过程中我们使用模拟数据。对我们而言，模拟数据集的确切组成是已知的，这可以帮助我们更简单地评估每个转录组组装软件捕获的基因和转录本的结构和数量。为了进行评估，我们构建了两个具有不同特征和不同reads分布的数据集。首先使用Flux Simulator网页上提供的定向人类RNA-seq协议中指定的确切参数生成第一个数据集(Sim-I)，然后再使用参数化的正态分布（N(250,20)）替换Sim-I中的实验性片段大小分布生成第二个数据集（Sim-II）。数据集Sim-I的fragment大小遵循基于Illumina序列的经验分布，数据集Sim-II的fragment大小遵循平均值为250、标准差为20的正态分布。

为了描述转录组组装软件的准确性，我们认为当预测转录本的内含子的边界被完全匹配或者第一个外显子的起点与真实的起点之间的距离小于100bp并且最后一个外显子的终点与真实的终点之间的距离小于100bp时，该转录本被正确识别。对于多外显子转录本而言，我们认为只有当它们的链也正确识别、链特异性RNA-seq 数据被使用并且单外显子转录本也被分配到正确的链上时，多外显子转录本才被正确组装。我们定义程序的敏感性（sensitivity）为在模拟实验中表达的转录本中被正确识别的转录本的数量，定义程序的精度（precision）为能够匹配真实转录本的预测转录本或转录本片段占预测转录本的百分比。在模拟数据集上的实验表明，StringTie无论是在数据集Sim-I上，还是在数据集Sim-II上，其敏感度和精确度都高于其他转录组组装软件，Cufflinks的表现仅次于StringTie。在模拟数据上，StringTie能够正确转录出7559个转录本，比Cufflinks多转录20%的转录本。实际上，在真实数据上StringTie也都比其他转录组组装软件拥有更高的敏感性和精度。比如，基于人血的90000000个reads，StringTie能够正确组装出10990个转录本，第二好的组装软件Cufflinks只能正确组装出7187个转录本。并且StringTie相比其他转录软件而言，其运行更快、占用空间更小。

# 第2章 转录组组装总体流程设计

## 2.1转录组组装两大模块

### 2.1.1 构建splicing graph

在本文中我们采用的是双末端测序的模拟数据，我们将长度固定的reads集合作为程序的输入。这些reads的长度相对来说比较小，大约在50-150bp，我们需要根据这些reads构建splicing graph以确定所有可能的转录本并对这些转录本进行定量分析。构建splicing graph的方式有两种，一种是不参考基因组的从头组装（de novo），一种是依赖于参考基因组的组装。

本文采用从头组装的方式，利用指定大小的k-mer进行构图，其中k-mer是指从read中截取的长度为k的子串。首先创建k-mer字典，即将reads数据转化为指定大小的k-mer，并存放至HashMap(kmer\_hash)中。将kmer\_hash中的k-mer按照在reads中出现的次数（该次数定义为k-mer的覆盖度）进行降序排序，其中的每一个k-mer都可以作为种子k-mer（seed\_kmer）。首先从目前未被使用的覆盖度最高的seed\_kmer开始进行正向延伸，将seed\_kmer的序列作为初始trunk（树干）序列。如果kmer\_hash中存在未被使用的k-mer，且该k-mer前k-1长度的子串与seed\_kmer的后k-1长度的子串序列相同（定义该k-mer为candidate），则将candidate的最后一个字符拼接到trunk的尾部。将candidate作为seed\_kmer重复以上操作，不断地寻找candidate并将candidate的最后一个字符拼接到trunk末尾，直至seed\_kmer找不到未被使用的candidate为止。同理，在正向延伸得到的trunk的基础上进行反向延伸。trunk序列的第一个k-mer作为seed\_kmer，如果kmer\_hash中存在未被使用的k-mer，且该k-mer后k-1长度的子串与seed\_kmer前k-1长度的子串序列相同（定义该k-mer为candidate），则将candidate的第一个字符拼接到trunk的头部。将candidate作为seed\_kmer重复以上操作，不断寻找seed\_kmer

的candidate并将candidate的第一个字符拼接到trunk头部，直至seed\_kmer找不到未被使用的candidate为止，即完成了反向延伸，得到了trunk序列。

trunk序列中可能存在某个k-mer，该k-mer还有未被使用的正向candidate或反向candidate，那么该k-mer即为一个分叉点。从分叉点开始继续按照上述步骤进行正向或反向延伸，延伸后的序列即可作为树杈从主干分叉出去形成分支，详细分叉过程见3.5.3和3.5.4。处理完所有的分叉点之后得到一个有向图，该有向图即为splicing graph。

### 2.1.2 转录本的鉴定与定量

由2.1.1得到了splicing graph，该图中的每一条路径都是一条可能的转录本，但是为了提高转录本重建的精度和敏感度，我们需要对这些转录本的丰度进行定量分析。计算splicing graph中的所有路径的权重（定义路径权重为k-mer per base，即平均每个碱基上覆盖的k-mer数），并按照路径权重对路径进行降序排序。依次从排好序的路径中取出权重最大的一条路径，为其创建一个流网络，流网络中的点与路径中的点一一对应。如果路径序列上的k-mer连接形成的片段（fragment）从点x开始且到点y结束，则连接x、y，并定义x、y两顶点之间边的权重为从x开始到y结束的fragment的数量。边的权重的具体计算过程详见3.6.3。同时，定义路径上每个顶点的权重为顶点序列上的k-mer连接形成的fragment数量，并按照3.6.1的计算方法计算每个顶点的权重。当顶点权重和边的权重都计算完毕，流网络构建完毕。

最后，在流网络的基础上运行最大流算法。先将流网络中的顶点V一分为二（分为Vin和Vout），连接Vin、Vout并设置Vin和Vout之间边的权重为顶点V的顶点权重即得到新的流网络。举例如图2-1所示：

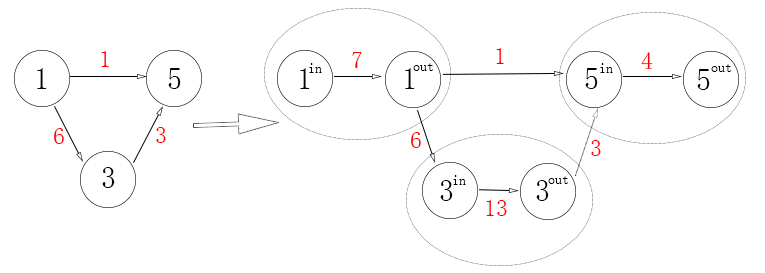


图 2-1 拆分流网络中的顶点

在新的流网络上运行最大流算法，如果流网络中存在增广路径，定义该增广路径上的流量（increment）为增广路径上所有边的权重的最小值，将流量增加到最大流（flow\_max）中并更新流网络中边的流量，直至找不到增广路径为止，得到的flow\_max即为该流网络的最大流。flow\_max大于0的路径代表的转录本即为本文的预测转录本，其转录丰度用flow\_max的大小表示。每找到一条预测转录本，该转录本使用的k-mer须在k-mer字典中更新其覆盖度，表示一定数量的该k-mer已被分配，然后继续处理下一条权重最大的路径，直至splicing graph中所有的路径都被处理完毕。转录组组装的总体流程图如图 2-2所示。

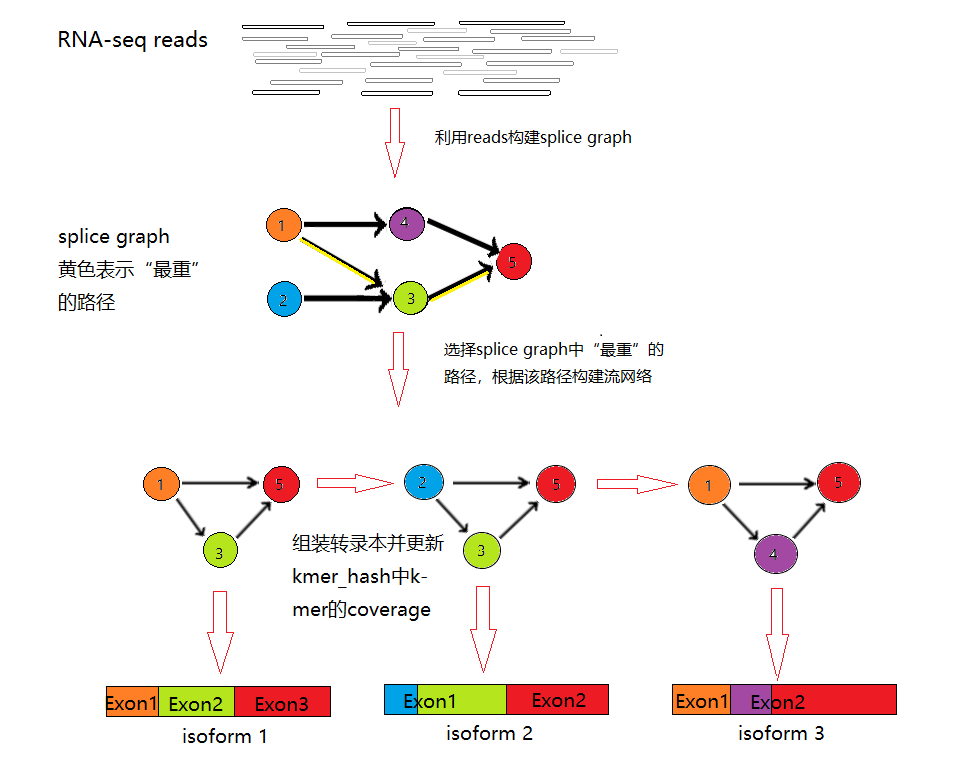


图 2-2 转录组组装总体流程图

# 第3章 转录组组装详细设计

## 3.1 数据结构设计

（1）read\_vector: Vector<String> (存放reads记录)

（2）kmer\_hash: Map<Long, Vector<Integer>> (存放k-mer,key值为k-mer的long型值，value值为k-mer所覆盖的reads的标号)

（3）kmer\_map: Map<Long, Integer>(存放k-mer，key值为k-mer的long型值，value值为k-mer所覆盖的reads的数目)

（4）list: List<Map.Entry<Long, Integer>> (存放排好序的k-mer)

（5）used\_kmers : Map<Long, Integer> (存放已被使用过的k-mer)

（6）bifurcations: Vector<Long> (存放trunk上的分叉点)

（7）path\_sum: Vector<Vector<Integer>>(存放splicing graph中的所有路径)

（8）Node: 自定义类，用于保存节点信息

## 3.2 图结构设计

为了更有效地对splicing graph上的顶点进行操作，本次设计采用自定义的Node类。Node类中一共有三个私有属性：children、parents、sequence。children的数据结构采用的是Vector<Integer>，里面存放其子节点的顶点标号。parents采用的也是Vector<Integer>，里面存放的是其父节点的顶点标号。sequence是String序列，保存该顶点的序列信息。为了保证其他类不会随意调用和修改Node类中的属性，在Node类里定义了一些公共接口以备调用。整个Node的类图如图 3-1所示。

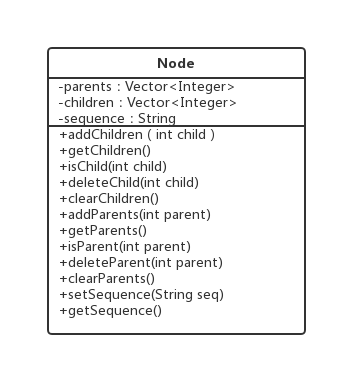


图 3-1 Node类图

### 3.2.1 addChildren(int child)

在该顶点的子节点集合children中加入节点child，使child成为该顶点的子节点。首先判断child是否合法，如果child<0或者该顶点的子节点集合中已经存在child，则直接返回false，添加子节点失败。否则将child加入该顶点的children集合中。addChildren方法如表3-2所示：

表3-2 addChildren

|  |
| --- |
| if (child < 0) {  return false;  } else {  if (children.size() != 0) {  for (int i = 0; i < children.size(); ++i) {  if (children.get(i) == child)  return false;  }  }  this.children.add(child);  return true;  } |

### 3.2.2 getChildren()

返回该顶点的子节点集合。return children;

### 3.2.3 isChild(int child)

判断child是不是该节点的一个子节点。首先判断child是否合法，如果child<0,直接返回false，child不是该节点的一个子节点。如果在该节点的children集合中找到了child，则说明child是该节点的一个子节点，返回true。方法isChild如表3-3所示：

表3-3 isChild

|  |
| --- |
| if (child < 0) {  return false;  } else {  for (int i = 0; i < children.size(); i++) {  if (children.get(i) == child) {  return true;  }  }  return false;  } |

### 3.2.4 deleteChild(int child)

删除该节点的子节点child。首先检查child是否合法，如果child<0，直接返回false。如果在该顶点的子节点中找到了child，将child移除，返回true。deleteChild方法如表3-4所示：

表3-4 deleteChild

|  |
| --- |
| if (child < 0) {  return false;  } else {  for (int i = 0; i < children.size(); i++) {  if (children.get(i) == child) {  children.remove(i);  return true; |

续表

|  |
| --- |
| }  }  return false;  } |

### 3.2.5 clearChildren()

将子节点集合清空。children.clear();

### 3.2.6 addParents(int parent)

在该顶点的父节点集合parents中加入父节点parent。首先判断parent是否合法，如果parent<0或者parent在该节点的parents集合中已经存在，则直接返回false，添加父节点失败。否则将parent加入该顶点的parents集合。addParents方法如表3-5所示：

表3-5 addParents

|  |
| --- |
| if (parent < 0) {  return false;  } else {  if (parents.size() != 0) {  for (int i = 0; i < parents.size(); ++i) {  if (parents.get(i) == parent)  return false;  }  }  this.parents.add(parent);  return true;  }  } |

### 3.2.7 getParents()

返回该顶点的父节点集合。return parents;

### 3.2.8 isParent(int parent)

判断parent是否为该顶点的一个父节点。首先判断parent是否合法，如果parent<0，直接返回false。如果在该顶点的父节点集合中找到了parent，则说明parent是该顶点的一个父节点，返回true。isParent方法如表3-6所示：

表3-6 isParent

|  |
| --- |
| if (parent < 0) {  return false;  } else {  for (int i = 0; i < parents.size(); i++) {  if (parents.get(i) == parent) {  return true;  }  }  return false;  } |

### 3.2.9 deleteParent(int parent)

删除该节点的父节点parent。首先检查parent是否合法，如果parent<0，直接返回false。如果在该顶点的父节点中找到了parent，将parent移除，返回true。deleteParent方法如表3-7所示：

表3-7 deleteParent

|  |
| --- |
| if (parent < 0) {  return false;  } else {  for (int i = 0; i < parents.size(); i++) {  if (parents.get(i) == parent) {  parents.remove(i);  return true;  }  }  return false;  } |

### 3.2.10 clearParents()

将父节点集合清空。parents.clear();

### 3.2.11 setSequence(String seq)

设置seq为当前顶点序列。sequence = seq;

### 3.2.12 getSequence()

返回当前顶点序列。return sequence;

## 3.3 基本操作设计

本文利用reads构建splicing graph，首先要根据reads创建k-mer字典，然后根据k-mer字典构建splicing graph。k-mer字典既要存储k-mer的值，又要存储该k-mer所覆盖的reads信息，因此存储k-mer时采用了HashMap这个数据结构。HashMap的key值应该存放k-mer的值，考虑到操作的简便性和空间复杂度，本次设计 用二进制值（64位，long）来代替String字符串存入HashMap。HashMap（kmer\_hash）的value值设定为Vector<Integer>，里面存放的是此k-mer覆盖的reads的集合。将k-mer字符串转化为long值存储，每两位可代表一个字符，即00代表‘A’，01代表‘C’，10代表‘G’，11代表‘T’。另外，对于k-mer的一些基本操作设计如下：

### 3.3.1 intToBase(int c)

定义char int\_to\_base[] = { 'A','C','G','T' }，根据输入的int c，返回int\_to\_base[c]即可得到相应的字符，即可将0、1、2、3转换为相应的A、C、G、T。

### 3.3.2 baseToInt(char base\_val)

根据输入的base\_val，判断其对应的数字，并将该数字返回。例如，base\_val为G，则返回2。

### 3.3.3 intvalToKmer(long int\_val,int kmer\_length)

将二进制的k-mer值（int\_val）转化为String形式。定义kmer=“”，每次取int\_val的最后两位，将其转化为相应的字符，将字符拼接在kmer的前面。然后int\_val右移两位，继续按照以上步骤进行处理，最终得到完整的k-mer字符串。intvalToKmer方法见表3-8：

表3-8 intvalToKmer

|  |
| --- |
| String kmer = "";  int c;  char base;  for (int i = kmer\_length-1;i >=0;i--) {  c = (int) (int\_val & 3);  base = *intToBase*(c);  kmer = base + kmer;  int\_val = int\_val >> 2;  } |

### 3.3.4 kmerToIntval(String kmer)

将输入的k-mer转化为二进制long值，设置初始intval=0,对于k-mer的每一位，得到其对应的int 值c，将intval右移，与c进行或运算，即可得到k-mer对应的二进制long值。kmerToIntval方法见表3-9：

表3-9 kmerToIntval

|  |
| --- |
| long intval = 0;  for (int i = 0;i < kmer.length();i++) {  int c=baseToInt(kmer.charAt(i));  intval =intval<< 2;  intval |= c;  } |

### 3.3.5 get\_forward\_candidates(long seed\_kmer, Map<Long, Vector<Integer>> kmer\_hash)

得到seed\_kmer的正向延伸候选k-mer（forward\_candidate）。如果kmer\_hash中存在一个k-mer，其前k-1长度的子串与seed\_kmer后k-1长度的子串序列相同，那么该k-mer即为seed\_kmer的一个forward\_candidate。由于k-mer的最后一位只有A、C、G、T四种可能，因此seed\_kmer的forward\_candidate的数量不会超过4。由于k-mer在kmer\_hash中作为long型进行存储，因此可以通过位移操作得到其forward\_candidate，如图 3-10。

将所有满足条件的forward\_candidate加入forward\_candidates（HashMap，key值为forward\_candidate的long值，value值为forward\_candidate的覆盖度），并对forward\_candidates中的数据按照value值进行降序排序以便每次在使用forward\_candidate的过程中都优先使用覆盖度最大的，最后将排序好的结果返回。

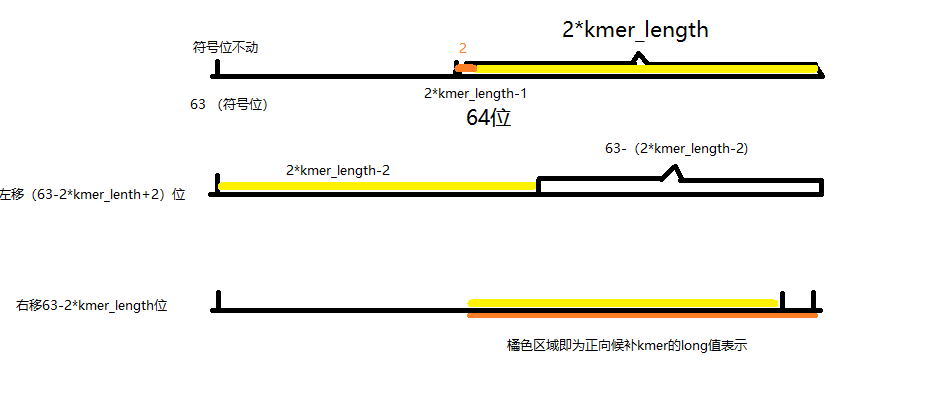


图 3-10 get\_forward\_candidates处理过程

get\_forward\_candidates的关键代码如表3-11所示：

表3-11 get\_forward\_candidates关键代码

|  |
| --- |
| long temp\_intval = seed\_kmer << (65 - kmer\_length \* 2) >> (63 - kmer\_length \* 2);  long a[] = { 0l, 1l, 2l, 3l };  for (int i = 0; i < 4; i++) {  long temp2 = temp\_intval;  temp2 |= a[i]; |

续表

|  |
| --- |
| temp2 = baseOptions.kmerToIntval(baseOptions.intvalToKmer(temp2, kmer\_length));  if (ifexist(kmer\_hash, temp2)) {  int read\_count = get\_readset\_count(kmer\_hash, temp2);  forward\_candidates.put(temp2, read\_count);  }  } |

### 3.3.6 get\_reverse\_candidates(long seed\_kmer, Map<Long, Vector<Integer>> kmer\_hash)

得到seed\_kmer的反向延伸候选k-mer（reverse\_candidate）。如果在kmer\_hash中存在一个k-mer，其后k-1长度的子串与seed\_kmer前k-1长度的子串序列相同，那么该k-mer即为seed\_kmer的一个reverse\_candidate。reverse\_candidate的第一位只有A、C、G、T四种可能，因此seed\_kmer的reverse\_candidate的数量也不会超过4。通过位移操作得到所有可能的reverse\_candidate（如图3-12），如果reverse\_candidate在kmer\_hash中存在，则将该reverse\_candidate加入reverse\_candidates（HashMap），同样按照reverse\_candidate的覆盖度进行降序排序作为结果返回。

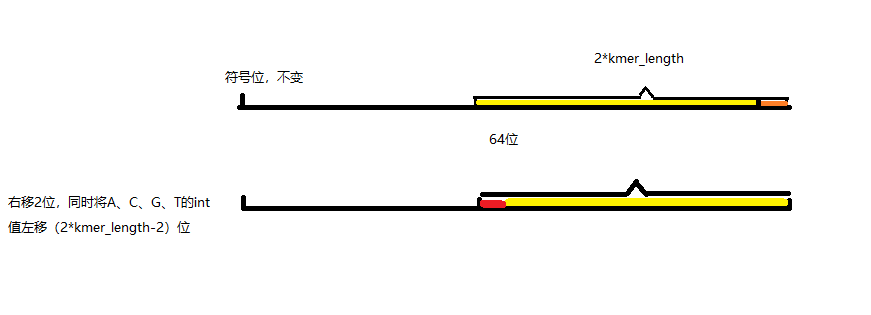


图 3-12 get\_reverse\_candidates处理过程

get\_reverse\_candidates的关键代码见表3-13：

表3-13 get\_reverse\_candidates关键代码

|  |
| --- |
| long temp\_intval = seed\_kmer >> 2;  long a[] = { 0l, 1l, 2l, 3l };  for (int i = 0; i < 4; i++) {  long temp2 = (a[i] << (2 \* kmer\_length - 2)) | temp\_intval; |

续表

|  |
| --- |
| temp2 = baseOptions.*kmerToIntval*(baseOptions.*intvalToKmer*(temp2, kmer\_length));  if (ifexist(kmer\_hash, temp2)) {  int read\_count = get\_readset\_count(kmer\_hash, temp2);  reverse\_candidates.put(temp2, read\_count);  }  } |

## 3.4 读取输入数据，构建HashMap

### 3.4.1 提取输入文件中的reads信息

输入文件应该有两个，分别保存测序两端的reads信息，例如sim55bp\_1.fa和sim55bp\_2.fa。先将sim55bp\_1.fa中的reads信息读取出来，放在read\_vector中，同时，每个read在read\_vector中的位置表示着read的标号，随着对输入文件的解析，标号依次递增，再将sim55bp\_2.fa中的reads取到read\_vector中。输入文件解析完成之后，read\_vector中保存了输入文件中所有的reads集合，每个read都对应着一个不重复的标号。

### 3.4.2 将提取的reads信息构建成HashMap

遍历read\_vector中的所有reads数据，根据输入的kmer\_length（默认为25）截取k-mer，将k-mer放入kmer\_hash中。kmer\_hash的key值存放的是k-mer转化为二进制之后的数值，其value值中存放的是Vector（readset），readset中存放该k-mer覆盖的reads的标号。每得到一个k-mer，先要判断在kmer\_hash中该k-mer是否已经存在，如果已经存在，那么将此时k-mer所在的read标号加入readset中。如果不存在，将该k-mer的long值作为key，创建一个包含该k-mer所在read编号的Vector作为value，将key和value加入kmer\_hash。当read\_vector中的每个read处理完毕，kmer\_hash构建完成。

## 3.5 根据kmer\_hash构建splicing graph

### 3.5.1 对kmer\_hash中的k-mer进行排序

kmer\_hash中的k-mer都是唯一的，在构建splicing graph之前，先对kmer\_hash中的数据按照其k-mer覆盖度进行排序。即按照在reads中出现的次数从大到小进行排序，将排序之后的结果存放在list中。

### 3.5.2 构建树干

**3.5.2.1 seed\_kmer正向、反向延伸**

按照顺序取出list中的k-mer，即目前未被使用的覆盖度最大的k-mer，将该k-mer作为seed\_kmer进行延伸。首先进行正向延伸，从seed\_kmer开始，如果有未被使用过的forward\_candidate,则表明可以进行延伸并利用forward\_candidate进行延伸扩展，然后将该forward\_candidate作为seed\_kmer一直延伸下去，直至seed\_kmer没有forward\_candidates或者seed\_kmer的forward\_candidates都已经被使用过。以同样的方式反向延伸，即可得到树干的序列,如果树干序列长度<200，则该树干长度不合理，选取list中下一个未被使用的seed\_kmer进行以上操作。

3.5.2.2 利用paired\_end reads信息正向、反向延伸

由于我们的输入数据是paired\_end reads，因此我们还需要根据paired\_end reads信息对树干进行扩展。paired\_end测序有两种方式：fr-firststrand和fr-secondstrand。fr-firststrand，第二个read与转录本方向一致，第一个read与转录本反向互补 （2-> <-1）。与fr-firststrand相反，fr-secondstrand第一个read与转录本方向一致，第二个与转录本反向互补（1-> <-2）,本次设计使用的模拟数据采用fr-secondstrand测序方式。

取树干的最后两个kmer\_length长度的字符串，把该字符串作为检查序列（check），检测是否有对应的paired\_end reads可以与其拼接。具体来说，对于字符串check中的每一个k-mer，我们从kmer\_hash中得到该k-mer对应的value值，此value值为标记k-mer覆盖的reads标号（id）的Vector,再把Vector中的数据取出放在reads(Vector)中。这样，当check中所有的k-mer处理完之后，reads中就存放了check对应的reads的标号，对这些reads的标号进行一一处理。从reads中依次取出read的id，根据paired\_end reads的测序方式，找到其对应的paired\_end read的id（paired\_end\_read\_id）。例如，一共有8个reads记录，分别是0、1、2、3、4、5、6、7，check对应的reads编号为0、5，那么其对应的paired\_end read的标号为4、1。

正向延伸：如果是fr-secondstrand方式，此时第一个read与转录本方向一致，于是可以根据第一个read的id找到第二个read的id（paired\_end\_read\_id），并判断第二个read反向延伸之后是否可以与第一个read进行拼接。

具体来讲，根据paired\_end\_read\_id从read\_vector中取出对应的read（paired\_end\_read），对于paired\_end\_read中的每一个k-mer，判断其是否被使用过，如果所有的k-mer都未被使用过，那么说明我们可以将此paired\_end\_read与check进行拼接。如果可以进行拼接，则在paired\_end\_read中找到覆盖度最大的k-mer（max\_read\_kmer），将max\_read\_kmer作为延伸的起点对其进行反向延伸。延伸过程中，维护一个HashMap(Map<Long, Boolean> use\_kmers)，use\_kmers中存放的是在此次扩展中被使用的k-mer。从max\_read\_kmer开始反向延伸，直至找不到reverse\_candidates或者所有的reverse\_candidates都已经在use\_kmers中，或者找到的reverse\_candidate为stop\_kmer(树干的最后一个k-mer)，停止延伸。将反向延伸得到的字符串放入StringBuffer（str）中，延伸过程中的最后一个k-mer作为extend\_kmer，我们在check中寻找extend\_kmer，如果找不到，说明扩展失败，则直接处理reads中的下一个read\_id，直至所有的read\_id都处理完。如果在check中找到了extend\_kmer，则说明trunk是可以再进行拼接的，记录匹配到的起始位置为start，此时修改trunk的序列得到新的trunk序列：

node\_set.get(node\_index).setSequence( node\_set.get(node\_index).getSequence().substring(0, start\_pos + start) + str)。

并将str中用到的k-mer都加入used\_kmers。取此时trunk序列的最后一个k-mer，再进行forward\_extend得到新的trunk序列。将新得到的序列重复执行上述操作，直至trunk无法进行正向扩展为止。整个过程如图 3-14所示。

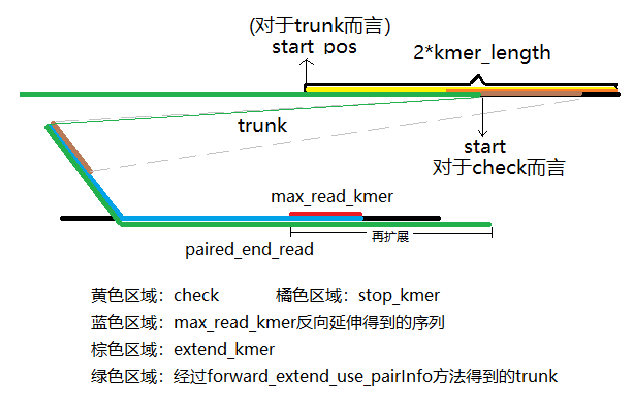


图 3-14 forward\_extend\_use\_pairInfo处理过程

反向延伸：在利用paired\_end reads进行反向延伸时，寻找对应的read的标号的处理与正向延伸有所不同.例如fr-secondstrand,我们应根据第二个read的id确定第一个read的id，将第一个read正向延伸与trunk进行拼接：

当找到对应的read（paired\_end\_read）之后，处理过程基本同正向延伸，不过是将trunk的前2\*kmer\_length长度的序列作为check，第一个k-mer作为stop\_kmer,从paired\_end\_read开始正向延伸，直至无法延伸或者最后一个延伸到的kmer为stop\_kmer为止。如果可以拼接，进行拼接，将拼接之后的trunk再进行反向延伸，得到最后的trunk。

### 3.5.3 正向构图

我们把3.5.2得到的每一个trunk都作为一个顶点，以该顶点为基础进行正向的构图和扩展，检查是否有分支，如果存在分支则需对顶点进行拆分，得到初始的splicing graph。

具体来讲，检查trunk的每一个k-mer，判断该k-mer是否有未被使用的forward\_candidates，如果有，将这些未被使用的forward\_candidates加入bifurcations（Vector<Long>）。如果bifurcations中的元素个数大于0，说明有可以分叉的地方，进行分叉操作。依次取出并删除bifurcations的最后一个元素（bifurcation\_val），利用未被使用的forward\_candidates对其进行正向延伸，直至不能再进行延伸为止。得到延伸后的序列（str），并将延伸过称中遇到的分叉点保存下来，如果延伸过程中的第一个分叉点是bifurcation\_val的话，那么说明bifurcation\_val还可以继续分叉，则再将bifurcation\_val添加到bifurcations中。定义str的最后一个k-mer为end\_kmer（long值定义为end\_intval），判断end\_kmer是否有forward\_candidates，如果没有，则说明str肯定是从树干分出去的树杈，不会再回到树干，则进行add\_branch操作。如果end\_kmer有candidates，则需判断trunk中是否可以找到其candidates，如果可以找到，则说明该分支最终连回树干，进行add\_bubble操作，否则进行add\_branch操作。

3.5.3.1 add\_branch(int node\_p, String branch, kmerHash kh)

node\_p表示顶点的编号， branch表示分叉的序列，为上面提到的str。

定义start\_kmer为branch的第一个k-mer，去树干中搜索start\_kmer，并记录找到的开始位置（start），如果找不到，start=-1，返回-1。由于在分叉的过程中会出现将bifurcations中的分叉点拆分开的可能，因此会出现start=-1的情况。首先判断node\_p是否需要拆分，如果start+kmer\_length-1小于顶点node\_p的序列长度的话，需要将node\_p一分为二，截取node\_p的start+kmer\_length之后的序列作为node1，剩下的作为node\_p。如果node1的序列长度符合要求，则将node\_p的子节点赋给node1，将node\_p的子节点清空,将node1添加到node\_p的子节点中 。将branch作为node2，并把node2添加到node\_p的子节点中，并截取node2的kmer\_length之后的序列更新node2。处理过程见图3-15，图中黄色区域表示可能的分叉点，从最后一个分叉点开始处理，橘黄色区域为最后一个分叉点进行正向延伸后的序列，蓝色表示node\_p，绿色表示node2，橘色表示node1：

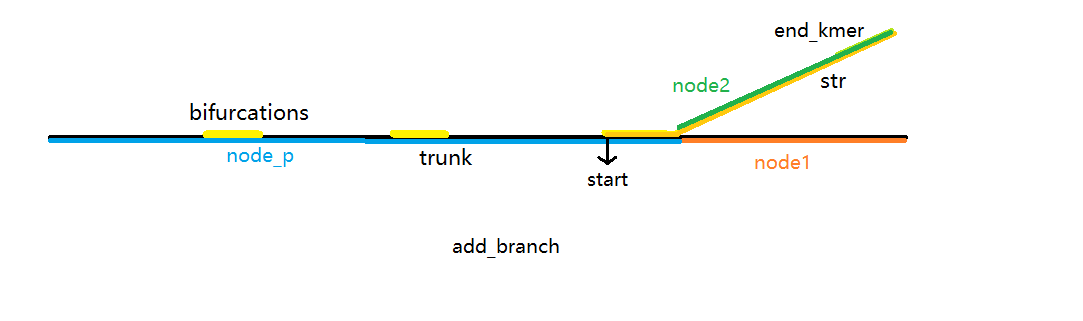


图3-15 add\_branch

3.5.3.2 add\_bubble (int node\_p, String str, kmerHash kh)

我们将str的第一个k-mer定义为anchor\_left，判断node\_p中是否可以找到anchor\_left，如果找不到，则说明在正向检查的过程中，anchor\_left可能已经被拆分，则不能进行分叉。如果找到了，定义anchor\_right为str的最后kmer\_length-1位。在目前已有的顶点中寻找anchor\_right,如果某个顶点序列中含有anchor\_right，则记该顶点为node\_q。如果找不到，则表明str只能作为树枝分叉出去，则进行add\_branch操作。如果找到了node\_q，定义如下变量（见表3-16）：

表3-16 定义变量

|  |
| --- |
| int anchor\_left\_length = anchor\_left.length();  int anchor\_right\_length = anchor\_right.length();  int end = node\_set.get(node\_q).sequence.indexOf(anchor\_right); |

同时根据str头尾连接处的coverage判断coverage是否足够支撑连接，如果coverage足够支撑连接，定义length为str.length() - anchor\_left\_length - anchor\_right\_length。

（1）node\_p=node\_q(str连接到node\_p本身)

定义distance 为end - start - anchor\_left\_length，对distance和length进行判断，出现以下情况（见表3-17）则返回，不做处理：

表3-17 不做处理的情况

|  |
| --- |
| if (distance > 0 && length == distance&& is\_similar(  node\_set.get(node\_p).sequence.substring(start + anchor\_left\_length, start + anchor\_left\_length + distance),  str.substring(anchor\_left\_length, anchor\_left\_length + length), 'F')) {  return -1;  } |

续表

|  |
| --- |
| if (length <= 0 && distance <= 0) {  return -1;  } |

否则，如果length<0，将anchor\_left\_length设置为anchor\_left\_length+length，如果distance<0，将anchor\_left\_length设置为anchor\_left\_length+distance，保证anchor\_left\_length与anchor\_right\_length加起来刚好等于str的长度，便于节点的拆分。如果length和distance都大于0，则不做处理。

截取node\_p在end位置之后的序列作为node1，将node\_p的子节点全部赋给node1，将node\_p的子节点清空。

如果distance<0(length>0):设置node\_p的节点序列为从第0个位置到第start+anchor\_left\_length（不包括第start+anchor\_left\_length位置上的字符）个位置的序列，并将node1加入node\_p的子节点集合。设置node2序列为str.substring(anchor\_left\_length, anchor\_left\_length + length - distance)，将node1加入node2的子节点集合，将node2加入node\_p的子节点集合。

如果distance>0(length<0):设置node3序列为node\_set.get(node\_p).sequence.substring(start + anchor\_left\_length,

start + anchor\_left\_length + distance - length)，将node\_p的子节点赋给node3，将node\_p的子节点清空，将node3加入node\_p的子节点集合,将node1加入node3和node\_p的子节点集合。

如果distance>0(length>0):设置node3序列为node\_set.get(node\_p).sequence.substring(start + anchor\_left\_length,

start + anchor\_left\_length + distance)，设置node2序列为 str.substring(anchor\_left\_length, anchor\_left\_length + length)，设置node\_p序列为node\_set.get(node\_p).sequence.substring(0, start + anchor\_left\_length)，将node\_p的子节点赋给node3，将node\_p的子节点清空，将node3和node2加入node\_p的子节点集合，将node1加入node3和node2的子节点集合。add\_bubble的过程如图图3-18所示，图中深绿色代表node\_p，浅绿色代表node1，红色代表node2，蓝色代表node3。

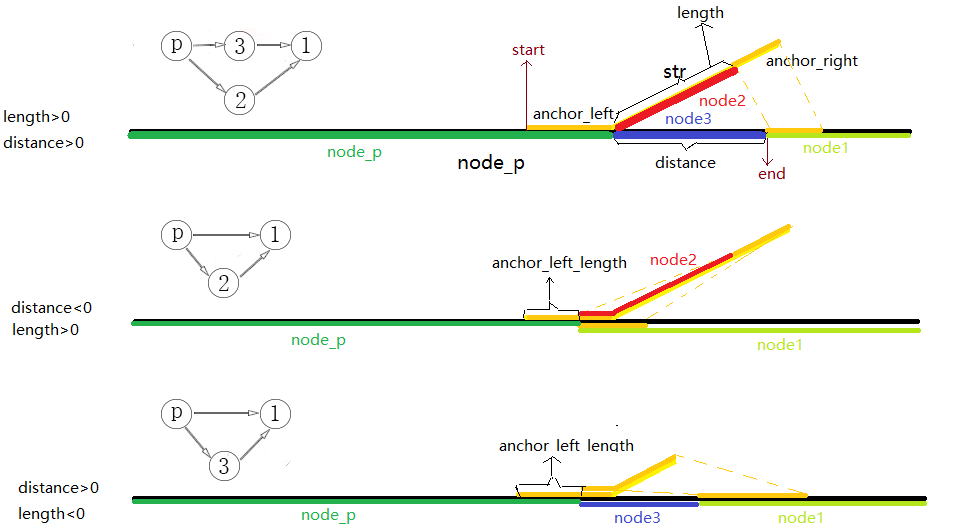


图 3-18 node\_p=node\_q

（2）node\_p!=node\_q（str连接到node\_q）

首先处理anchor\_left\_length，如果length<0，设置 anchor\_left\_length = anchor\_left\_length + length。如果start + anchor\_left\_length小于node\_p的序列长度，则进行以下操作：设置node1为node\_set.get(node\_p).sequence.substring(start + anchor\_left\_length)，设置node\_p为node\_set.get(node\_p).sequence.substring(0, start + anchor\_left\_length)，将node\_p的子节点赋给node1，将node\_p的子节点清空，将node1加入node\_p的子节点集合。

如果length>0（如图3-19所示），设置node2为str.substring(anchor\_left\_length, anchor\_left\_length + length)，否则设置node2等于node\_p。如果end等于0，则直接将node\_q加入node2的子节点集合，否则需要设置新的节点分割node\_q。当需要分割时，设置node3为node\_set.get(node\_q).sequence.substring(end)，将node\_q的子节点赋给node3，将node\_q的子节点清空，将node3加入node\_q和node2的子节点集合。设置node\_q的序列为node\_set.get(node\_q).sequence.substring(0, end)。

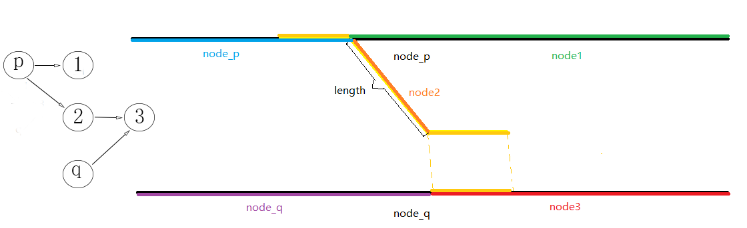


图 3-19 node\_p!=node\_q

### 3.5.4 反向构图

在正向构图之后可能会出现很多个顶点，在这些顶点的基础上，对每一个顶点进行反向构图。该方法表示为reverse\_check\_and\_extend(kmerHash kh, int node\_index)，其中node\_index表示顶点的标号。对于顶点node\_index的顶点序列中的每一个k-mer，检查其是否有reverse\_candidates，如果有且还未被使用过，就将reverse\_candidate加入bifurcations（Vector<Long>）。处理完每个k-mer之后，如果bifurcations中的元素个数不为0，则依次从末尾取出bifurcations中的分叉点（intval），以intval为seed\_kmer进行反向延伸,直至不能再延伸为止，得到延伸后的字符串str。定义end\_kmer为str的第一个k-mer，得到end\_kmer的reverse\_candidates，如果reverse\_candidates的数量大于0，则进行add\_reverse\_bubble操作，否则进行add\_reverse\_branch操作。由于目前splicing graph中使用的k-mer都是不可再扩展的，否则构成的图是不完全的，因此如果forward\_candidates的数量大于0，说明end\_kmer至少有一个forward\_candidate，既然是end\_kmer，则end\_kmer是无法再连接下去的，因此其candidates一定被使用过，在各个顶点中寻找end\_kmer，如果找到某个顶点node\_q含有end\_kmer，则str的头部可以拼接到node\_q上，如果找不到，说明顶点在拆分的过程中可能恰好将end\_kmer拆分开，此时不做处理。

3.5.4.1 add\_reverse\_bubble

定义anchor\_right为str的最后一个k-mer，在node\_p的序列中查询anchor\_right并记录找到的开始位置（start），如果不能找到，start=-1，返回。定义anchor\_left为str的第一个k-mer。在所有的已知顶点中查找anchor\_left，如果没有顶点包含anchor\_left或者该顶点是node\_p本身，则不做处理。如果找到了不是node\_p的顶点，记录该顶点为node\_q。定义anchor\_length为anchor\_left的长度，end为node\_q中end\_kmer所在的开始位置。

如果end+anchor\_length等于node\_q的序列长度（如果小于，则说明anchor\_left也存在两个或两个以上的forward\_candidates，说明此时正向检查没有做完，是不合理的。），定义length为str.length() - anchor\_length - kh.kmer\_length。

如果length>0，设置node1为str.substring(anchor\_length, length + anchor\_length)，将node1加入node\_q的子节点集合。如果length<0，直接将node1定义为node\_q。如果start=0，则直接将node\_p加入node1的子节点集合，否则定义新节点node2，其顶点序列为node\_set.get(node\_p).sequence.substring(start)，将node2加入node1的子节点集合，并将node\_p的子节点赋给node2，将node\_p的子节点清空，将node2加入node\_p的子节点集合，同时，设置node\_p的顶点序列为node\_set.get(node\_p).sequence.substring(0, start)。处理过程见图 3-20，其中node\_q节点序列不变，蓝色表示新增节点node1，node\_p拆分为浅绿色节点node\_p和深绿色节点node2。

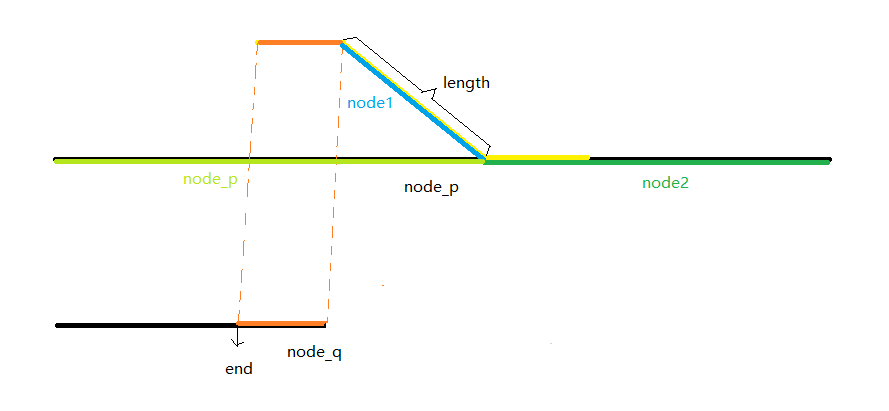


图 3-20 add\_reverse\_bubble

3.5.4.2 add\_reverse\_branch

定义end\_kmer为str的最后一个k-mer，str\_length 为str.length() - kh.kmer\_length，start为node\_p中出现end\_kmer的开始位置，如果start=-1，直接返回。设置node1为node\_set.get(node\_p).sequence.substring(start)，设置node2为str.substring(0, str\_length)。如果start等于0，直接将node\_p加入node2的子节点集合，否则设置node\_p为node\_set.get(node\_p).sequence.substring(0, start)。将node\_p的子节点赋给node1，将node\_p的子节点清空，将node1加入node\_p和node2的子节点集合，当start不为0时，拆分过程见图3-21，其中橘色表示node1，蓝色表示node2，绿色表示node。

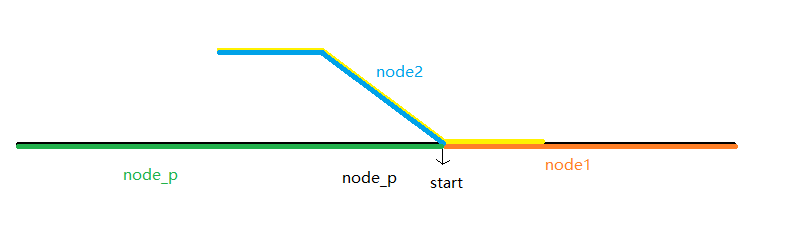


图 3-21 add\_reverse\_branch(start!=0)

### 3.5.5 重置父节点

因为在正向构图和反向构图的过程中，只是进行了子节点的修改，因此最后统一重置父节点。重置父节点代码见表3-22：

表3-22 重置父节点

|  |
| --- |
| for (int i = 0; i < node\_set.size(); i++) {  Vector<Integer> children = new Vector<Integer>();  children = node\_set.get(i).getChildren();  for (int j = 0; j < children.size(); j++) {  node\_set.get(children.get(j)).addParents(i);  }  } |

至此，splicing graph构建完毕。

## 3.6 根据splicing graph运行最大流算法

### 3.6.1 计算每个顶点的权重

如何计算顶点的coverage是本次设计的一大难点，经过多日的思考与演算，找到了正确的解决办法。对于一个顶点p，依次得到p的序列上的所有k-mer，并根据kmer\_map得到k-mer的覆盖度（覆盖reads的数目，记为count），并将count加入node\_kmer\_count（Vector<Integer>）。如果某个顶点的序列长度小于kmer\_length，则遍历其子节点和父节点，如果有一个子节点或者父节点可以和该顶点拼接为一个k-mer大小的顶点，则设置该顶点的count为拼接得到的k-mer的coverage。在node\_kmer\_count寻找不被0分割的连续区域，将该区域中元素的最小值累加到顶点的权重中，并将该区域内所有元素都减去最小值以更新区域元素，递归地处理不被0分割的区域，直至区域区域只有一个元素时，将该元素累加至顶点权重，返回。

举例说明：例如一个顶点的node\_kmer\_count为[5,4,6,10,3,3,10,5,6,7,9]，首先我们对该node\_kmer\_count进行排序，得到[3,3,4,5,5,6,6,7,9,10,10]。则第一个数字的大小代表了目前得到的顶点厚度（局部厚度不同的顶点，厚度取决于最薄的地方），也就是说，目前有三条片段可以完整地覆盖此顶点。将node\_kmer\_count中的每个元素都减3，得到新的node\_kmer\_count。当node\_kmer\_count中有数为0时，那么由此位置开始的k-mer两边是不能连接的，因为没有足够的kmer支持。于是采用一分为二的思想分别计算左右两边，直至只剩下一个数。最终计算得到该顶点的coverage，将结果保存在node\_cov数组中。如果splicing graph中只有一个顶点，那么该顶点即可作为一个转录本，其转录丰度为开始时node\_kmer\_count中最小的数。上述例子的计算见图3-23。

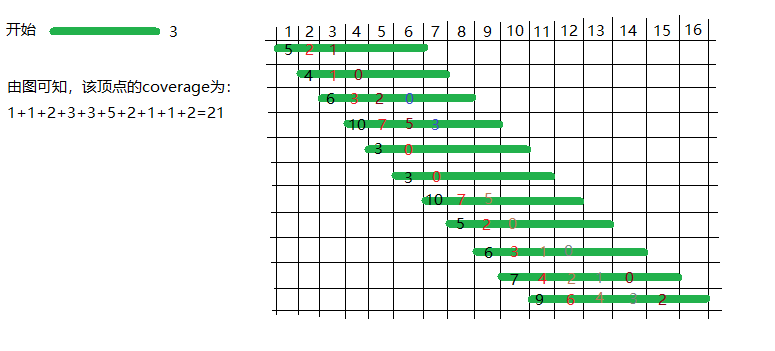


图 3-23 计算顶点的coverage

### 3.6.2 对路径权重进行排序

我们定义最重的路径为k-mer per base coverage（平均每个碱基上覆盖的k-mer最多）最大的路径。首先，根据父子节点的关系找出源点和汇点集，其代码见表3-24：

表3-24 找出所有源点和汇点

|  |
| --- |
| for (int k = 0; k < sg.node\_set.size(); k++) {  if (sg.node\_set.get(k).getChildren().size() == 0) {  des.add(k);  }  if (sg.node\_set.get(k).getParents().size() == 0) {  src.add(k);  }  } |

根据源点和汇点利用dfs找出所有的路径（见表3-25），将得到的路径加入path\_sum(Vector<Vector<Integer>>)中：

表3-25 dfs寻找所有路径

|  |
| --- |
| path.add(src);  if (src == des) {  Vector<Integer> path\_c = new Vector<Integer>();  for (int i = 0; i < path.size(); i++) {  path\_c.add(path.get(i));  }  path\_sum.add(path\_c);  return;  }  for (int i = 0; i < edges.length; i++) {  if (edges[src][i] != 0) {  dfs(edges, i, des, kh, f, path\_sum);  path.remove(path.size() - 1);  }  }  } |

对于path\_sum中存放的所有的路径，计算该路径上所有k-mer的覆盖度之和，用此覆盖度除以路径长度即可得到路径的权重，并根据路径权重对每条路径进行倒序排序，将排序的结果放在list\_path中。

### 3.6.3 计算流网络中边的权重

在3.6.1中每个顶点的coverage已经计算出，现在需要计算边的权重。我们定义，如果有一个fragment从x开始，到y结束，那么我们就定义x到y有一条边，该边的权重为连接x、y两点的fragment的数量。如何计算fragment的数量，是本次设计的又一难点。

首先，记录路径为path\_highest（Vector<Integer>），我们将该路径上的顶点序列都拼接在一起，组成一个长str（path\_str）同时记录path\_str的长度，并将每个顶点序列的边界保存在terminal中（path\_str和terminal的计算方法见表3-26）：

表3-26 计算path\_str和terminal

|  |
| --- |
| String path\_str = ""; |

续表

|  |
| --- |
| for (int i = 0; i < path.size(); i++) {  length += node\_set.get(path.get(i)).getSequence().length();  path\_str += node\_set.get(path.get(i)).getSequence();  terminal.add(length - 1);  } |

将path\_str中的每一个k-mer的覆盖度大小加入到kmer\_counts（Vector<Integer>）中，kmer\_counts的每个位置都代表了每个k-mer的起始标号。在kmer\_counts中寻找不被0分割的长度大于1的连续区域，记录该区域两端的编号，并根据terminal和两端编号判断该区域连接连接的两个顶点的标号，如果两个顶点编号不同，则该区域中所有数字的最小值即为这两个顶点的边的权重。如果该连续区域长度等于1，判断该位置的k-mer是否横跨两个顶点，如果是则将该位置的数字累加到这两个顶点的边的权重上。每处理完一个连续区域，将该区域上的所有元素都减去刚才计算得到的权重，表示权重数量的k-mer已被使用。采用分而治之的思想寻找所有满足上述要求的连续区域，判断区域连接的顶点，并不断按照上述方法增加两顶点之间的权重，直到所有不被0分割的区域都处理完毕。将所有边的权重保存在二维数组edges\_ter中。

下面用简单的例子来演示我们的计算过程，设路径上有3个顶点，这三个顶点的编号分别为0、2、1，顶点序列分别为ACTGATGCCT、TTTACTGG、CCAGATCGAACT，设kmer\_length为6。按照表3-26所示代码可计算出path\_str为ACTGATGCCTTTTACTGGCCAGATCGAACT，terminal为[9,17,20]，length为30。

假设按照path\_str中每个k-mer的覆盖度计算得到的kmer\_counts为[5,4,6,10,3,3,10,5,6,7,9,4,4,7,9,8,6,6,7,6,6,9,4,3,4]，用start和end分别表示kmer\_counts中不被0分割的连续区域的起始标号和结束标号，用start\_index和end\_index表示起始顶点和结束顶点分别是路径上的第几个点。开始时，start=0，end=24，处理kmer\_counts中位置0到位置24之间的区域。首先计算start位置代表的k-mer第一个字符所在的顶点编号start\_index，再计算end位置代表的k-mer最后一个字符所在的顶点编号end\_index，start\_index和end\_index仅为顺序编号，并不是顶点的真实编号（计算过程见表3-27）。计算得start\_index=0，end\_index=2，因此对应的起始顶点真实编号为0，结束顶点真实编号为1。

表3-27 计算start\_index和end\_index

|  |
| --- |
| for (int i = 0; i < terminal.size(); i++) {  if (start < terminal.get(i)) {  start\_index = i;  break;  }  }  for (int j = terminal.size() - 1; j >= 0; j--) {  if (end + kh.kmer\_length - 1 > terminal.get(j)) {  end\_index = j + 1;  break;  }  } |

然后计算区域中的最小数值作为边的权重。kmer\_counts排序后最小的数为3，则顶点0到顶点1的边覆盖的fragment的数量为3（权重为3），即fragment\_count[0][1]=3。然后将start和end之间的区域中所有元素都减3， [2,1,3,7,0,0,7,2,3,4,6,1,1,4,6,5,3,3,4,3,3,6,1,0,1,0]为更新后的数据。

在kmer\_counts后面人工添加一个0，便于后面计算。用count\_is\_zero(Vector)记录kmer\_counts中为0的元素的位置，则count\_is\_zero为[4,5,23,25]。0元素将kmer\_counts分割成三个不为零的连续区域，先处理第一个区域[2,1,3,7]，此时start=0，end=3。由于3+6（kmer\_length）刚好等于terminal[0](9)，因此该区域无法连接两个顶点，因此该区域不做处理。处理第二个区域[7,2,3,4,6,1,1,4,6,5,3,3,4,3,3,6,1]，此时start=6，end=22，按照表3-27计算可知start\_index=0，end\_index=2，则对应的实际顶点标号为0、1，该区域最小值为1，则有fragment\_count[0][1]=3+1=4。将该区域的所有元素减去1得到[6,1,2,3,5,0,0,3,5,4,2,2,3,2,2,5,0]，其中有两个不被0分割的连续区域，则依次处理区域[6,1,2,3,5]和[3,5,4,2,2,3,2,2,5],重复按照以上方法递归地进行计算。当整个第二个区域处理完之后，接着处理第三个区域,当所有区域处理完时边的权重已计算完成。余下的计算结果如图3-28所示，其中图中绿色的线表示顶点最后一个k-mer即将开始，红色的线表示顶点最后一个字符已结束，黄色标记的数字表示新增的边的权重。

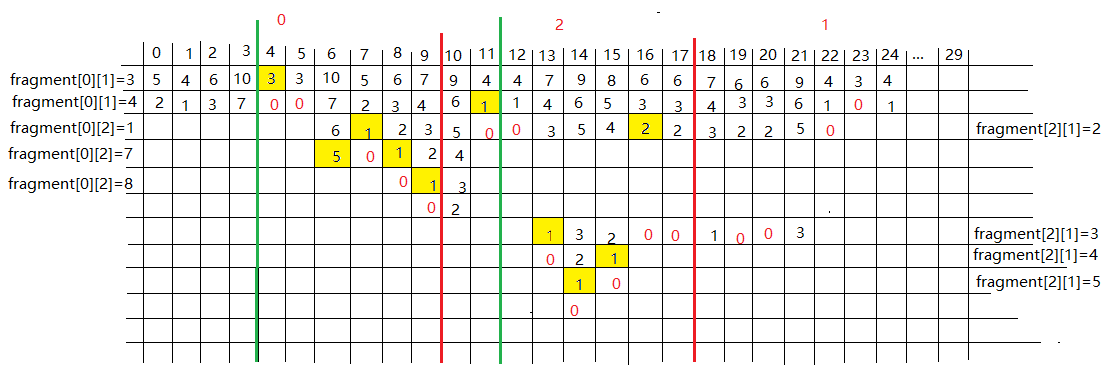


图 3-28 计算边的权重示例

计算边的权重关键代码如表3-29所示：

表3-29 计算边的权重

|  |
| --- |
| for (int i = 0; i < count\_is\_zero.size(); i++) {  int cer = start;  if (i == count\_is\_zero.size() - 1) {  end\_biao = cer + count\_is\_zero.get(i) - 1;  Vector<Integer> kmer\_counts\_c = new Vector<Integer>();  for (int j = start\_biao - cer; j <= end\_biao - cer; j++) {  kmer\_counts\_c.add(kmer\_counts.get(j));  }  compute\_fragment(kmer\_counts\_c, start\_biao, end\_biao, path, terminal, fragment\_count, kh);  break;  }  if (count\_is\_zero.get(i) == 0) {  start\_biao++;  continue;  }  int index = count\_is\_zero.get(i);  if (index + cer == start\_biao) {  start\_biao++;  continue;  }  int index\_index = 0;  int start\_pos = 0;  for (int j = 0; j < terminal.size(); j++) {  if (start\_biao < terminal.get(j)) { |

续表

|  |
| --- |
| start\_pos = j;  break;  }  }  boolean flag1 = false;  for (int j = terminal.size() - 1; j >= 0; j--) {  if (index + cer - 1 + kh.kmer\_length - 1 > terminal.get(j)) {  index\_index = j + 1;  flag1 = true;  break;  }  }  if (flag1 == false) {  index\_index = 0;  }  if (index+cer - 1 + kh.kmer\_length - 1 < terminal.get(index\_index) && index\_index == start\_pos) {  start\_biao = cer+index + 1;  continue;  }  if (start\_biao + index - 1 + kh.kmer\_length - 1 == terminal.get(index\_index) && index\_index == start\_pos) {  start\_biao = start\_biao + index + 1;  continue;  }  end\_biao = cer + index - 1;  Vector<Integer> kmer\_counts\_c = new Vector<Integer>();  for (int j = start\_biao - cer; j <= end\_biao - cer; j++) {  kmer\_counts\_c.add(kmer\_counts.get(j));  }  compute\_fragment(kmer\_counts\_c, start\_biao, end\_biao, path, terminal, fragment\_count, kh);  } |

### 3.6.4 重构流网络运行最大流算法

首先根据edges\_ter和node\_cov信息重新构图，将path\_highest中的顶点一分为二。比如splicing graph中一共有12个顶点，从0开始标号，path\_highest为0->4->2->1->6，根据edges\_ter构建的初始流网络如图3-30所示。在初始流网络中，如果两个顶点之间有边，则表示有fragment从其中一个顶点开始，到其中一个顶点结束，边上的权重即为fragments的数量。

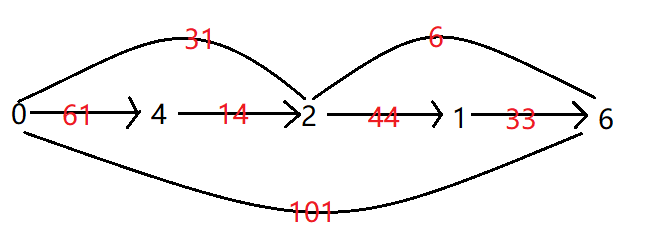


图 3-30初始流网络

将图中的顶点一分为二，新增的顶点的标号即为原始顶点的标号加上splicing graph中顶点的个数，新增顶点与原始顶点之间的边的权重为原始顶点的顶点权重，则新构图如图3-31：

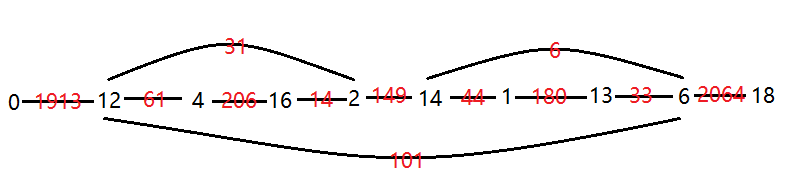


图 3-31 重构流网络

与此同时，我们试图增加偏差因数来平衡顶点和与顶点连接的边之间的覆盖度的差距，因此我们定义偏差因数为（1-1）。fxy表示的是从x开始到y结束的fragment，w(fxy)表示为fxy的重量，定义w(fxy)=1/n，n为fxy的数量。bv其实代表的是到v结束的所有顶点的权重的倒数之和除以从v开始的所有顶点的权重的倒数之和（包括自己到自己）。我们可以根据之前求得的edges\_ter和node\_cov计算顶点的偏差因数。由于我们已经计算出edges\_ter和node\_cov，一个顶点v，其node\_cov减去其与子节点、父节点的边的权重，则剩下fvv的数量，则可根据（1-1）式计算出每个顶点的bv。我们在重构流网络时，将顶点V分为了两个顶点Vin和Vout并连接这两个点，并把V的coverage作为边的权重。因此我们在图3-31的基础上，在每条Vin到Vout的边上都乘以bv,即得到了最终的流网络。综上所述重构图的代码如表3-32所示：

表3-32 重构图

|  |
| --- |
| for(int i=0;i<edges.length;i++){  new\_edge[i][i+edges.length]=node[i]\*bv[i];  for(int j=0;j<edges.length;j++){  if(edges[i][j]!=0){  new\_edge[i+edges.length][j]=edges[i][j];  }  }  } |

然后运行最大流算法。第一步，寻找增广路径。由于覆盖度越高的转录本表明其在现实中出现的概率越大，因此在寻找增广路径的过程中，我们按照边的权重的大小进行寻找，即首先寻找权重最大的增广路径，并用prefix记录增广路上所有点的前驱。

找到增广路之后，计算这条路径上的流量，每当得到一条增广路上的流量之后，我们要在kmer\_hash中将使用到的k-mer的覆盖度减去流量，表明这些k-mer已被分配，下次不可再用。当流网络中找不到增广路时，最大流算法停止，得到该预测转录本的转录丰度，处理下一个转录本，这样即完成了转录本的定量分析。 最大流算法伪代码如表3-33所示：

表3-33 最大流算法伪代码

|  |
| --- |
| Input: flow network with sink, source, new\_edge  Output: maximum flow flow\_max  Initialization: set flowmax=0  while there is an augmenting path p in network  set u to sink (the last node in p)  increment = ∞  while there is predecessor v of node u in p  increment = min(increment,new\_edge[v][u])  u = v  end while  for all consecutive nodes u,v (u before v) in p  new\_edge[v][u]-=increment  new\_edge[u][v]+=increment  end for  flow\_max += increment  end while |

当splicing graph中所有的路径都处理完时，最大流不为0的路径是我们预测的转录本。

# 第4章 改进与总结

## 4.1 结果分析

在这里我们定义该程序的精度为正确组装的转录本条数占预测转录本总数百分比，定义召回率为正确组装的转录本条数占真正转录本的百分比。此次的测试在模拟数据上进行，首先运行算法得到预测转录本，并给每一个转录本一个可以区分的编号，例如将预测转录本输出到sim55bp.fasta文件，文件中的每条转录本都有特定编号。在linux环境下下载安装blast和trinityrnaseq-2.2.0，安装完成之后，对参考转录本文件refTranscripts.fasta（其中共有100条参考转录本）进行建库：

makeblastdb -in refTranscripts.fasta -out refTranscripts -dbtype nucl

将预测转录本与参考转录本进行对比：

blastn -query sim55bp.fasta -db refTranscripts -out blastn.fmt6.txt

-evalue 1e-20 -dust no -task megablast -num\_threads 2 -max\_target\_seqs 1 -outfmt 6

执行trinityrnaseq-2.2.0中的analyze\_blastPlus\_topHit\_coverage.pl脚本：

perl analyze\_blastPlus\_topHit\_coverage.pl blastn.fmt6.txt refTranscripts.fasta sim55bp.fasta

对不同bp长度的数据运行程序，当kmer\_length为25时，结果如表4-1：

表4-1 结果分析

| bp长度 | 预测转录本条数 | 匹配率>=95%的转录本数量 | 精度 | 召回率 |
| --- | --- | --- | --- | --- |
| 55 | 123 | 12 | 0.0976 | 0.12 |
| 60 | 124 | 9 | 0.0726 | 0.09 |
| 65 | 135 | 9 | 0.0667 | 0.09 |
| 75 | 121 | 11 | 0.0910 | 0.11 |
| 80 | 131 | 13 | 0.0992 | 0.13 |
| 85 | 157 | 14 | 0.0891 | 0.14 |
| 90 | 138 | 15 | 0.1087 | 0.15 |
| 95 | 151 | 16 | 0.1060 | 0.16 |
| 100 | 144 | 17 | 0.1181 | 0.17 |
| 105 | 153 | 19 | 0.1242 | 0.19 |
| 110 | 164 | 21 | 0.1280 | 0.21 |
| 115 | 153 | 25 | 0.1634 | 0.25 |
| 120 | 150 | 21 | 0.1400 | 0.21 |
| 125 | 122 | 23 | 0.1885 | 0.23 |
| 130 | 132 | 27 | 0.2045 | 0.27 |

针对以上表格数据绘制折线图，根据折线图分析可得，当k-mer长度一定时，该算法随着reads长度的增大，召回率和精度总体上呈上升趋势，reads的长度越大，召回率和精度相对就越高。当reads长度为130bp时，召回率最大，当reads长度为130bp时，精度最大，如图4-1。

图 4-1 不同bp长度的reads数据比较结果

正确预测的转录本条数趋势与召回率趋势相同，总体呈上升状态，在130bp时最大（见图4-2）。

图 4-2 正确组装的转录本的数目

当reads长度为130bp时，对于k-mer长度不同的数据也得到了不同的结果，其结果见表4-3：

表4-3 130bp时不同k-mer长度的结果分析

| k-mer长度 | 预测转录本条数 | 匹配率>=95%的转录本数量 | 精度 | 召回率 |
| --- | --- | --- | --- | --- |
| 25 | 132 | 27 | 0.2045 | 0.27 |
| 26 | 111 | 26 | 0.2342 | 0.26 |
| 27 | 135 | 25 | 0.1852 | 0.25 |
| 28 | 110 | 26 | 0.2363 | 0.26 |
| 29 | 114 | 28 | 0.2456 | 0.28 |
| 30 | 116 | 28 | 0.2414 | 0.28 |
| 31 | 123 | 26 | 0.2114 | 0.26 |

根据以上数据绘制折线图，观察折线图发现随着k-mer长度的增加，召回率较为平稳，在k-mer长度为30和29的时候召回率最大。如图4-4。

图 4-4 相同reads长度，不同k-mer长度时的结果分析

## 4.2 改进

我们初始版本的算法中加入了偏差因数，但是是否在模拟数据上不加偏差因数结果可能会更好是一个值得验证的问题，我们去掉偏差因数对算法进行了改进，当k-mer长度为25时，改进后的算法结果分析如表4-5所示：

表4-5 改进后结果分析

| bp长度 | 预测转录本条数 | 匹配率>=95%的转录本数量 | 精度 | 召回率 |
| --- | --- | --- | --- | --- |
| 55 | 135 | 12 | 0.0889 | 0.12 |
| 60 | 112 | 8 | 0.0714 | 0.08 |
| 65 | 102 | 9 | 0.0882 | 0.09 |
| 75 | 124 | 12 | 0.0968 | 0.12 |
| 80 | 112 | 13 | 0.1161 | 0.13 |
| 85 | 178 | 18 | 0.1011 | 0.18 |
| 90 | 149 | 18 | 0.1208 | 0.18 |
| 95 | 205 | 20 | 0.0976 | 0.20 |
| 100 | 160 | 21 | 0.1313 | 0.21 |
| 105 | 132 | 17 | 0.1288 | 0.17 |
| 110 | 185 | 22 | 0.1189 | 0.22 |
| 115 | 179 | 26 | 0.1453 | 0.26 |
| 120 | 160 | 26 | 0.1625 | 0.26 |
| 125 | 160 | 27 | 0.1688 | 0.27 |
| 130 | 192 | 30 | 0.1563 | 0.30 |

对上表进行折线图分析可知，当k-mer长度一定时（25），精度和召回率随着reads数据长度的增加总体呈上升趋势，当reads长度为130bp时，召回率最大，当reads长度为125bp时，精度最大，结果如图4-6所示。

图 4-6 改进后不同bp长度的reads数据比较结果

当reads长度为130bp时，不同的k-mer大小得到不同的结果，其结果见表4-7：

表4-7 改进后130bp的reads数据在不同k-mer长度下的结果分析

| k-mer长度 | 预测转录本条数 | 匹配率>=95%的转录本数量 | 精度 | 召回率 |
| --- | --- | --- | --- | --- |
| 25 | 192 | 30 | 0.1563 | 0.30 |
| 26 | 170 | 27 | 0.1588 | 0.27 |
| 27 | 144 | 27 | 0.1875 | 0.27 |
| 28 | 151 | 29 | 0.1921 | 0.29 |
| 29 | 146 | 31 | 0.2123 | 0.31 |
| 30 | 127 | 28 | 0.2205 | 0.28 |
| 31 | 158 | 29 | 0.1835 | 0.29 |

根据以上数据绘制折线图，观察折线图发现随着k-mer长度的增加，精度呈逐渐上升趋势，召回率较为平稳，在k-mer长度为29的时候召回率最大，k-mer长度为30时精度最高，如图4-8。

图 4-8 改进后相同reads长度，不同k-mer长度时的结果分析

将改进前后运算结果的精度和召回率进行比较发现，改进之后的算法在召回率上要比改进之前要好一些，改进之后的精度的上升趋势也更加平缓，但是改进之后的算法其最大精度要比改进之前小。经过分析我们认为，当程序中加入偏差因数时，在构造流网络的过程中对边的流量加上了相关限制，再求最大流的过程中流量计算更加严格，产生的转录本的精度相对来说要比没有偏差因数的时候要高一些。具体的比较结果如图4-9所示：

图 4-9 改进前后结果对比

## 4.3 总结

综上，本文设计和实现了基于流分解的转录组组装，首先利用reads信息构造了splicing graph，然后为splicing graph上的每一条路径构造流网络以计算表达丰度。我们在模拟数据上实现了该算法，实验结果表明随着reads长度的增加和k-mer长度的增加，精度和召回率总体上呈上升趋势。考虑到模拟数据相对于真实数据来说要小很多，因此在构建流网络去掉了原有的偏差因数进而进行了改进，改进的结果表明在模拟数据上召回率有所提高，精度上升趋势更加平缓。现有的研究结果表明，基于该算法的转录组组装软件召回率最高能达到50%以上，而本文实现的该算法召回率仅能达到30%以上。总的来说，目前本文的算法其精度和召回率都还比较低，还有很大的改进空间，希望后续可以有一些好的改进可以提高精度和召回率。