东北大学秦皇岛分校

Northeastern University at Qinhuangdao

毕业论文

蛋白质同源性搜索的高效算法

院　　别	计算机与通信工程学院
专业名称	计算机科学与技术
班级学号	**1803 - 20188117**
学生姓名	项溢馨
指导教师	王和兴

2022 年 6 月

郑 重 声 明

本人呈交的学位论文，是在导师的指导下，独立进行研究工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确的方式标明。本学位论文的知识产权归属于培养单位。

本人签名：　　日期：

蛋白质同源性搜索的高效算法

摘　　要

传

关键词：蛋白质同源性搜索，计算生物学算法，最小哈希算法，局部敏感哈希算法

An Efficient Algorithm for Protein Homology Search

Abstract

Traditional .

Key words: Protein Homology Search, Algorithms in computational biology, MinHash, Locality-sensitive hashing

目　　录

1 绪　论

1.1 引言

如果两个或多个结构由一个共同的祖先演化而来，则称其同源 (Homology) 。在生物信息学中，同源主要是指序列上的同源，用来说明两个或多个蛋白质或 DNA 序列具有相同的祖先。

1.2 研究背景

检索同源蛋白质序列是蛋白质生物信息学的基本问题，通常是任何基于序列的蛋白质研究的第一步，也是信息量最大的步骤之一。

例如，由 DeepMind 团队开发的人工智能软件系统 AlphaFold 能够根据一个蛋白质的氨基酸序列来确定它的 3D 结构，对于准确认识蛋白质功能，破解蛋白质折叠难题有着至关重要的意义。AlphaFold 会先在已有的蛋白质序列和结构数据库里面寻找目标蛋白质的同源蛋白，构成神经网络的输入[1]。AlphaFold 的预测精度依赖于同源蛋白的数量和相似性，以及同源蛋白是否已经有实验结构。

蛋白质的同源性常常通过序列的相似性 (Sequence similarity) 来判定，相似性一般用检测序列和目标序列之间序列一致性 (Percent identity) 来表示。当两个序列的相似性超过偶然预期时，我们推断同源性。对序列过度相似的最简单解释是，这两个序列不是独立出现的，而是来自一个共同的祖先。共同祖先解释了过度的相似性（其他的解释要求相似的结构独立出现）。因此，过度的相似性意味着共同的祖先[2]。

然而，同源性和相似性的概念不能等价。同源序列并不总是具有显著的序列相似性。有数以千计的同源蛋白质序列相似性比对并不显著。但当相似性搜索发现具有统计意义的匹配时，我们可以推断这两个序列是同源的。

所以同源检索的研究目标是在海量蛋白质序列中挖掘出相似性满足一定要求的序列集合，作为下游蛋白质结构预测与功能分析任务的关键输入。同源蛋白质序列检索抽象为数学问题就是：在超大数据量的字符串中，检索出相似度高于某个阈值的字符串对，并且一般情况下相似度用 Levenshtein 距离（编辑距离）度量。

由于测序技术的飞速发展，被测序蛋白质的数量在迅速增加。现阶段该问题的研究难点在于序列数目过于庞大（超过10^9），序列间两两比较的朴素算法的$O(n^2)$时间复杂度是不可行的，更何况两个序列间计算 Levenshtein 距离还需要更多的计算。因此，在合

理的时间内找到同源蛋白质序列是本文要解决的关键问题，也是难点所在。

1.2.1 国内外研究现状

序列比对是生物信息学最重要的方法之一。通过序列比对可以阐明序列之间的相似性程度，从而确定目标之间的亲缘关系和结构关系[3]。序列比对也可分为双序列比对和多序列比对。双序列比对技术较为成熟。主流的基础序列比对算法有 Smith-Waterman 算法[4]和 Needleman-Wunsch 算法[5]等。通过启发式提速，被广泛使用的双序列比对软件有 FASTA 和 BLAST 软件。多序列比对是双序列比对的扩展，更为复杂，目前尚无比较有效的算法，其中较流行的软件是 CLUSTALW。

尽管有许多算法和工程上的改进，但两个序列之间的序列比对或编辑距离的计算仍是输入序列长度的平方级别，这在实践中仍然耗时巨大，甚至不可接受。基于此，近来部分算法依靠哈希来降维，以便更快速地检测具有高对齐概率的序列。MinHash 和 Locality-sensitive hashing (LSH) 方法被广泛运用在 Mash[6]，Mashmap[7]和 MHap[8]等序列比对工具中。

1.2.2 Needleman-Wunsch 算法及其优化

Needleman-Wunsch (NW) 算法是将动态规划应用于生物序列比较的最早期的几个实例之一，也被称为优化匹配算法和整体序列比较法。NW 算法能找到两个序列的最佳比对，是一种全局比对技术，但无法用于寻找高度相似的局部区域。该算法将两个序列之间的对应情况分为 Match，Mismatch 和 Indel（Insertion 或 Deletion）三类，并为之分配相应的得分。原始的动态转移方程为

$$F_{ij} = max\{F_{i-1,j-1} + S(A_i, B_j),\ F_{i,j-1} + d,\ F_{i-1,j} + d\ \} \tag{1.1}$$

A_i表示第一个序列的第 i 位，B_i表示第二个序列的第 i 位，F_{ij}表示从A_0和B_0更新到A_i和B_j的匹配得分情况。$S(A_i, B_j)$对应A_i和B_j Match 或 Mismatch 的得分，d对应 Indel（即在A_i或B_j相应位置插空匹配）的得分。

该算法对于计算两个长度分别为 m 和 n 的序列的时间复杂度为$O(mn)$，运用 Method of Four Russians 算法进行优化可将复杂度降为$O(mn/log_n)$[9][10]。

Rashed 等人提出了基于机器学习的 NW 算法优化，将包括多层感知机 (MLP) 、支持向量机 (SVM) 、决策树、随机梯度下降法 (SGD) 和 XGBoost 分类器在内的 15 种机器学习方法运用在原始数据的分类上，快速并行实现 NW 算法[11]。

1.2.3 Smith-Waterman 算法及其优化

Smith-Waterman (SW) 算法是 NW 算法的变体。该算法的目的不是进行全序列的比对，而是找出两个序列中具有高相似度的片段。相较于 NW 算法而言，SW 算法更新阶段不存在负数得分情况，这使得局部序列比对成为可能。任何位置的分数低于 0，意味着此前的序列不具备相似性。将之设置为 0，达到了“重设”的效果，使得从此位置开始的比对不受之前位置的影响。动态转移方程可以表示为

$$\mathrm{H}_{k0} = \mathrm{H}_{0l} = 0, (0 \le k \le n, 0 \le l \le m) \tag{1.2}$$

$$H_{i,j} = max \begin{cases} H_{i-1,j-1} + s(a_i, b_j), \\ max_{k \ge 1}\{H_{i-k,j} - W_k\}, \\ max_{l \ge 1}\{H_{i,j-l} - W_l\}, \\ 0 \end{cases} \tag{1.3}$$

SW 算法的时间复杂度与 NW 算法相类似，也是平方级别。FASTA 运用了 SW 算法，并将之并行化，使比较速度提升了 10-20 倍[12]。FASTA 算法通过搜索和匹配 k-tuples（长度为 k 的子序列）来近似最佳匹配。对于蛋白质序列来说，这些 k-tuples 的长度往往为 2。虽然 FASTA 算法的最坏时间复杂度仍为$O(mn)$，但平均时间复杂度可以缩小到$O(\frac{mn}{20^k})$[13]。因此，FASTA 算法的复杂性取决于 k-tuples 的大小：k-tuples 越大，算法的速度越快。尽管 FASTA 算法比之前任何的算法都要快，但它无法保证找到两个蛋白质序列之间的最佳比对。因为它使用 k-tuples 来加速，可能会错过较小的相似区域，从而可能使两个蛋白质错位配对。此外，限制动态规划搜索区域的范围可能会产生次优对齐。很明显，FASTA 算法在速度和精度之间需要权衡。

此外，通过 FPGA[14]、GPU[15]、SIMD[16]和 Cell Broadband Engine（Cell 宽带引擎）[17]的加速，都很好地提升了 SW 算法的性能，但在最坏情况下仍是平方级别的复杂度。

1.3 研究意义与内容

2000 年 6 月 26 日，生物学和医学经历了历史性的巨变。英国首相布莱尔和美国总统克林顿宣布完成人类基因组草案。当时的报道宣称“科学家破解了人类生命的遗传密码”。人类基因组计划的进展以及多种生物基因测序工作的完成，标志着现代生物学的一个新开端。其中大多数生物学和生物医学研究将以“基于序列”的方式进行。大量 DNA 序列和蛋白质序列已被测定，人类跨入了蛋白质时代。

人类基因组序列只是已知的许多完整基因组序列中的一种。广泛分布在生命树分支

中的生物体的基因组序列，给了我们一种地球上所有生命都具有强大的统一性的感觉。人类基因组本质上是信息。计算机对于序列的确定以及生物学和医学应用都是必不可少的。计算机不仅提供了处理和存储数据的原始能力，还提供了实现结果所需的复杂数学方法。生物学和计算机科学的结合创造了生物信息学。

生物信息学数据的一个明显特点是数据量非常大。目前，核苷酸序列数据库包含16×10^9个碱基，这仅相当于五个人类基因组序列的数量。大分子结构数据库包含 16000 个条目，即蛋白质的完整三维坐标，平均长度约为 400 个残基。不仅单个数据库很大，而且它们的规模正在以非常高的速度增长[18]。

多数计算机程序根据蛋白质的氨基酸序列决定其三维结构，从而决定其功能特性的基本原理，计算这些蛋白质的结构。第一步通常是数据库筛选已知结构的相关蛋白质，结构预测的问题将被简化为预测序列变化对结构的影响，并且目标结构将通过同源建模的方法进行预测。如果没有发现已知结构的同源蛋白质，那么结构预测必须完全从头开始。

因此，蛋白质同源性搜索，即序列相似性度量，是生物信息学中许多算法的核心。综上，如何将序列比对算法与生物信息学中的寻找同源蛋白质序列问题的特性相结合，进一步提高序列比对算法在求解同源蛋白质序列问题时的性能，已成为了一个值得关注的研究方向。围绕这一前沿研究领域，本文尝试开发一种快速算法来实现同源蛋白质序列的快速检索。

1.4 论文组织结构

第 1 章绪论，阐明了本文的研究背景，概括了该领域国内外主要的研究成果，阐明了本文做出的主要贡献，最后对于本文的工作内容跟进行了简明扼要的概括和整理。

第 2 章是蛋白质同源性搜索问题的综述。首先对同源性相关的生物学基础概念进行了介绍，并使用形式化语言对同源性搜索问题进行了定义，用编辑 (Levenshtein) 相似度来狭义化同源性概念，便于之后的模型建模与分析。最后，从如何度量相似性的角度，概括表述了解决蛋白质同源性搜索问题的总体思路，即从快速过滤实现序列规模的降维和计算 Levenshtein 距离的近似解两方面进行优化。

第 3 章旨在详述本文提出的蛋白质同源性搜索的高效算法。首先从最长公共子序列 LCS 问题入手，引出 Jaccard 相似度，证明 Jaccard 相似度是编辑相似度的上界，并利用该上界实现低同源性序列的快速过滤。接着从算法时间复杂度角度进行论证，寻找问题

规模降维的可行性。之后，引入 Order Min Hash 模型，先对文本相似度算法和散列函数交叉领域的术语做了详细介绍，分析 MinHash 算法和 Locality-sensitive hashing 算法的原理，并具体计算了概率和期望。对于 Order Min Hash 模型的理念，给出了基于 C++的实现方法。最后，基于 MinHash 算法和 Locality-sensitive hashing 算法，对 Order Min Hash 模型进行修改，给出 Jaccard 相似度的高效近似估计算法。

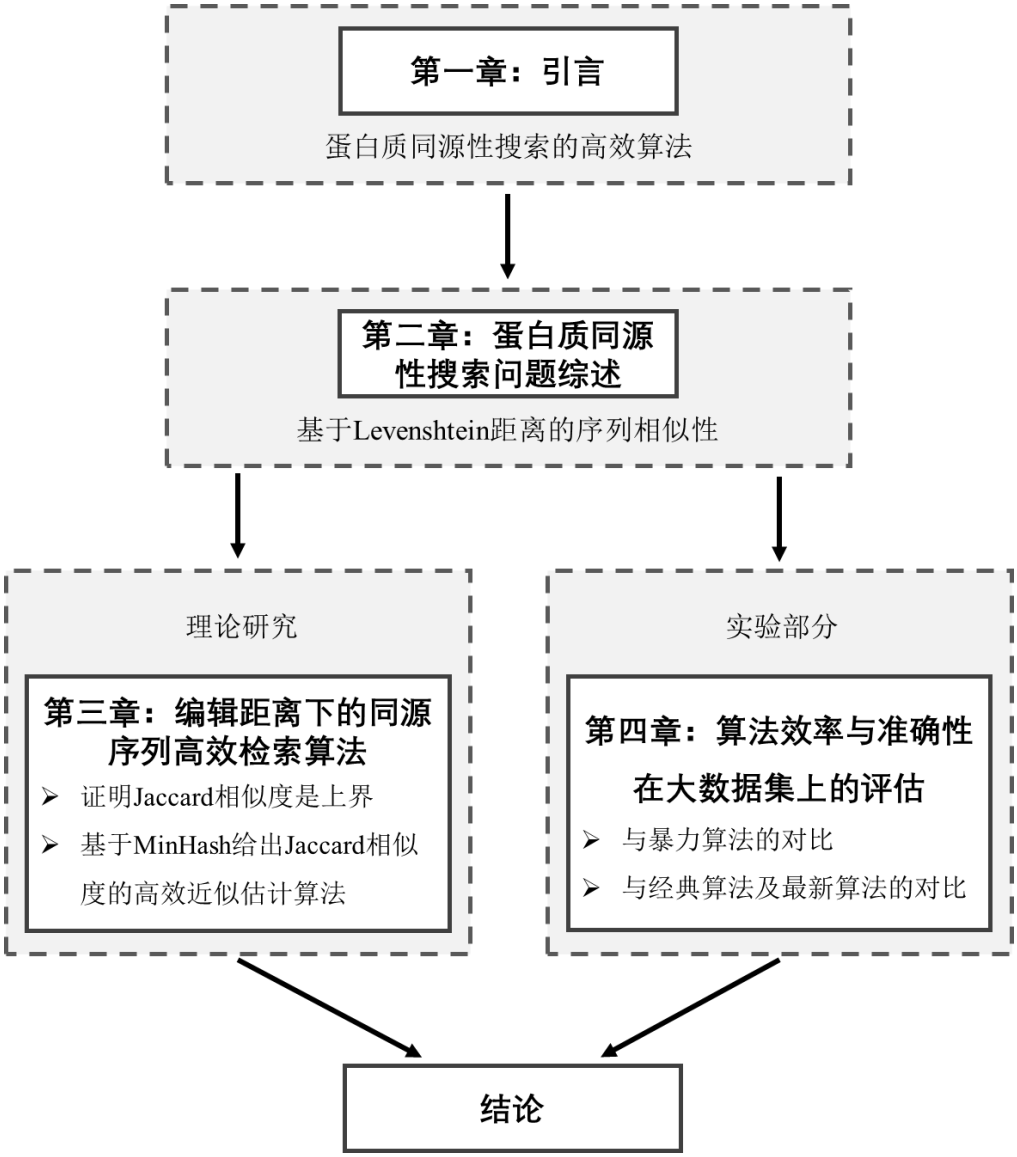

图 1.1　论文各章节的关系图

第 4 章属于实验环节，首先对数据集与将要进行的实验任务进行介绍，接着详述了实验设计中各类参数，包括实验环境、数据集预处理过程、模型参数设置与具体的训练流程，方便其他研究者复现。再将实验结果与朴素算法的结果进行比对分析，检测算法的可信程度和实现效率。并将本篇提出的模型算法进行多次实验，其结果与该领域最新

模型结果进行对比，取得了较高的性能。最终对模型的各项参数进行优化，并进行多组对照实验，找到最优的参数。实验证明，当 k=4，l=2，L=300，p=19260817，pp=500 时，模型能取得比较理想的效果。

结论是正文的最后部分，将在此对全文的工作进行总结，并对未来进一步研究工作的发展提出思路和展望。

2 蛋白质同源性搜索问题综述

同源性搜索的核心是要找到相似性较高的序列对，其中主要包含两个方面的难题需要解决。一方面是如何度量出这种相似性，另一方面是如何降低需要进行相似性分析的数据规模。

2.1 基本术语介绍

（1）脱氧核糖核酸 DNA

DNA 是一种聚合物，由两条多核苷酸链组成。它们相互缠绕，形成一个双螺旋结构，携带遗传指令。每个核苷酸由四种含氮核碱基，脱氧核糖和磷酸基团组成。在 DNA 分子中，脱氧核糖和磷酸是稳定的，而碱基是可变的。

DNA 分子中的碱基分为嘌呤和嘧啶。嘌呤包括腺嘌呤 (adenine) 和鸟嘌呤 (guanine) ，嘧啶包括胞嘧啶 (cytosine) 和胸腺嘧啶 (thymine) ，还有一种碱基是尿嘧啶 (uracil) 存在于 mRNA 中[错误!未找到引用源。]。图 2.1 列出了五种碱基的分子结构。在 DNA 分子中，腺嘌呤，鸟嘌呤，胞嘧啶和胸腺嘧啶分别用字母 A、G、C 和 T 表示。

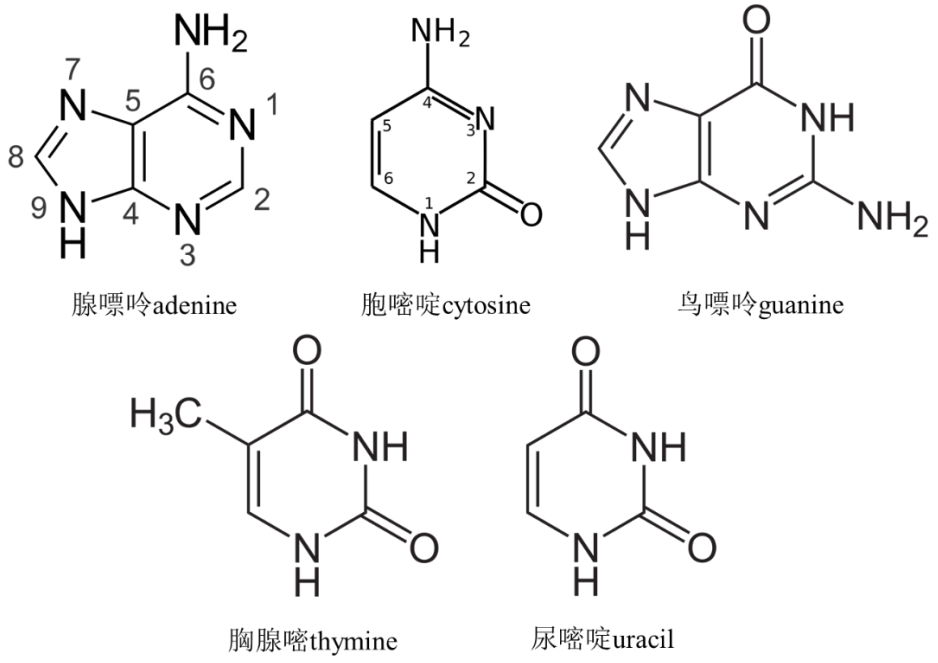

图 2.1　碱基的分子结构

（2）氨基酸 Amino acids

氨基酸是组成蛋白质的基本单位，赋予蛋白质特定的分子结构形态。如图 2.2 所示，

氨基酸由一个氨基、一个羧基、一个氢原子和一个 R 基连在同一个中心 C 原子上组成。

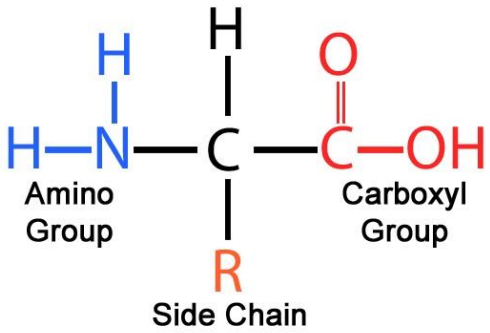

图 2.2　氨基酸的基本结构

氨基酸中的 R 基可以是不同侧链基团，对应的理化性质也不同。基因中每三个碱基序列组合表示一个氨基酸，例如 AUG 是蛋氨酸的序列，因此可能的密码子总数为 64 个。但是由于基因中存在一些冗余，一些氨基酸可以对应多个序列组合。

蛋白质氨基酸，即标准氨基酸，共 22 种。其中 20 种是常见的，见于所有生物体中。由于这 20 种氨基酸的结构和化学性质不同，形成了功能各异的蛋白质分子。

（3）蛋白质 Protein

蛋白质由氨基酸组装而成。每种蛋白质都有自己独特的氨基酸序列，由编码该蛋白质基因的碱基序列决定。蛋白质的一级结构即氨基酸序列。蛋白质是生物大分子，分子量一般在 8000 以上。按照氨基酸的平均分子量 110 计算，蛋白质所含氨基酸的数目为约为 73 个。图 2.3 展示了蛋白质是怎样由碱基通过转录组合构成的。

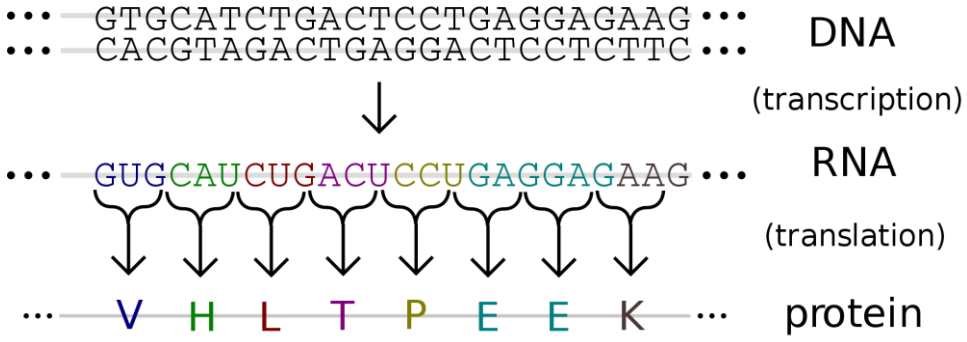

图 2.3　碱基与蛋白质序列关系

（4）同源性 Homology

在生物学中，同源性是指不同类群中的一对结构或基因之间由于拥有共同的祖先而产生的相似性。蛋白质的序列同源性同样根据共同祖先来定义。由于物种形成事件（直

系同源），片段重复事件（旁系同源）和基因横向转移事件（异源同源）的作用，两个不同的 DNA 片段可从共同祖先演化而来。

蛋白质之间的同源性一般由氨基酸序列的相似性推断。因为两个蛋白质之间具有同源性，因此它们的氨基酸序列高度相似，并且具有相似的结构和功能。

2.2 蛋白质同源性搜索问题定义

输入两条蛋白质序列：WLVAKRKOTAXHZHXKKPHLWYADZPHVSOKTXZELPPG 和 VFTRGPGFPBPXTGKPGGXPULAPGLAAGPMBL。直接对他们从头进行序列匹配的话，匹配数非常少，这两条序列并没有什么相似之处。但如果将第二条序列后移 2 位，就可以发现它们的相似性增加。如果进一步在第二条序列中插入一位空格，就会发现原来这两条序列有更多的相似之处。

图 2.4　蛋白质序列比对

上面是两条序列相似性的一种定性表示方法。为了说明两条序列的相似程度，还需要定量计算。有两种方法可用于量化两条序列的相似程度：一为相似度，它是两条序列的函数，其值越大，表示两条序列越相似；与相似度对应的另一个概念是两条序列之间的距离，距离越大，则两条序列的相似度就越小。在大多数情况下，相似度和距离可以交互使用，并且距离越大，相似度越小，反之亦然。但一般而言，相似度使用得较多，并且灵活多变。

相异度 (dissimilarity) 可以被定义为一个函数d: $\mathcal{U} \times \mathcal{U} \rightarrow [0,1]$，表示在全集$\mathcal{U}$内两个对象之间的距离。$d$必须符合一些基本条件，即：

（1）非负性：$d(x,y) \geq 0$；

（2）对称性：$d(x,y) = d(y,x)$；

（3）同一性：$d(x,x) = 0$。当$d(x,y) = 0$时，意味着 x=y；

（4）三角不等式：$d(x,z) + d(y,z) \geq d(x,y)$。

因此相异度是一种取值在[0,1]区间的归一化距离。相似度 (similarity) 定义为函数s，那么$1 - s$就是相异度。因此，相异度可以定义相似度，反之亦然。

最简单的距离就是海明距离 (Hamming distance, HD) 。对于两条长度相等的序列，海明距离等于对应位置字符不同的个数。例如，图 2.5 表示了 3 组序列海明距离的计算结果。

```
a=    EHX       SHHDB       AHBCRYEGFHF
b=    EXX       WIWDJ       ABCRYEQGFHF
-------------------------------------------------------------------
HD =   1          4                6
```

图 2.5 海明距离的计算

使用距离来计算不够灵活。因为序列可能具有不同的长度，两条序列中各位置上的字符并不一定是真正的对应关系。例如，在实际情况中，可能会发生像删除或插入一个氨基酸这样的错误。虽然两条序列的其他部分相同，但由于位置的移动导致海明距离的失真。就图 2.5 中最右边的情况，海明距离为 6。简单地从海明距离来看，两条序列差别很大（整个序列的长度只有 11），但是，如果从 a 中删除 H，从 b 中删除 Q，则两条序列都成为 ABCRYEGFHF，这说明两条序列仅仅相差两个字符。实际上，在许多情况下，直接运用海明距离来衡量两条序列的相似程度是不合理的。

为了解决字符插入和删除问题，引入字符“编辑操作” (Edit Operation) 的概念。通过编辑操作将一个序列转化为一个新序列。用一个新的字符“-”代表空位 (Space) ，并定义下述字符编辑操作：

Match (a, a) — 字符匹配；

Delete (a, -) — 从第一条序列删除一个字符，或在第二条序列相应的位置插入空白字符；

Replace (a, b) — 以第二条序列中的字符 b 替换第一条序列中的字符 a；

Insert (-, a) — 在第一条序列插入空位字符，或删除第二条序列中的对应字符 b。

很显然，在比较两条序列 s 和 t 时，在 s 中的一个删除操作等价于在 t 中对应位置上的一个插入操作，反之亦然。需要注意的是，两个空位字符不能匹配，因为这样的操

作没有意义。

给定两个字符序列 a 和 b，分别用$|a|$和$|b|$表示序列的长度。编辑距离 (Levenshtein distance) 定义为

$$\mathrm{lev(a,b)}=\begin{cases}|a| & if\ \ |b|=0,\\ |b| & if\ \ |a|=0,\\ \mathrm{lev}\big(\mathrm{tail(a)},\mathrm{tail(b)}\big) & if\ \ a[0]=b[0],\\ 1+min\begin{cases}\mathrm{lev(tail(a),b)}\\ \mathrm{lev(a,tail(b))}\\ \mathrm{lev(tail(a),tail(b))}\end{cases} & otherwise\end{cases} \tag{2.1}$$

tail(x)代表字符序列 x 除了第一个字符外的其余字符序列。最小值函数中的三种情况分别对应从 a 到 b 匹配的删除、插入和替换。

编辑相异度 (edit dissimilarity/ normalized edit distance) $E_d(a,b)$代表将 a 转换成 b 所需的编辑距离除以 a、b 中较长序列的长度，见公式 2.2。由此可以得到本文讨论的蛋白质同源性搜索问题的核心，编辑相似度$E_s(a,b)$。

$$E_d(a,b)=E_d(b,a)=\frac{lev(a,b)}{\max(|a|,|b|)} \tag{2.2}$$

$$E_s(a,b)=1-E_d(a,b) \tag{2.3}$$

2.3 总体思路

若采用朴素算法进行两两序列比对，时间复杂度$\mathrm{O}(l^2)$，l表示字符序列的数量。

而传统计算 Levenshtein 距离的算法时间复杂度为$\mathrm{O}(mn)$，m 和 n 分别表示进行比对的两条序列的长度。考虑其使用动态规划思想，无法简单通过改造算法结构来实现时间复杂度的降维。虽然可以通过$\mathrm{O}(\max(m,n))$的时间查询对应位置相同的字符并进行删除的预处理，缩小$\mathrm{O}(mn)$中的 m 和 n，使时间复杂度降为$\mathrm{O}(m'n')$ $(m'\le m,n'\le n)$ 。但由于蛋白质序列的特殊性，使得这种优化发挥空间的余地很小。

因此总的序列比对时间复杂度近似为$\mathrm{O}(l^2mn)$，是绝对无法接受的。整个改进算法最主要的提升仍然在对l的降维上。因此我们需要找寻一种数据清洗方式，对同源概率非常低的序列对进行快速过滤，使l的规模迅速缩小。并且最后能通过并行处理的方式，使得程序在多组测试数据上同时运行。

而对于相似度的计算过程进行降维处理，既然无法通过算法层面的优化来计算准确的 Levenshtein 距离，我们可以尝试寻找新的相似度计算方法去逼近 Levenshtein 距离的上下界，以可以接受的精确度误差来近似编辑相似度。

2.4 本章小结

本章首先对同源性相关的生物学基础概念进行了介绍，并使用形式化语言对同源性搜索问题进行了定义，用编辑相似度来狭义化同源性概念，便于之后的模型建模与分析。最后，从如何度量相似性的角度，概括表述了解决蛋白质同源性搜索问题的总体思路，即从快速过滤实现序列规模的降维和计算 Levenshtein 距离的近似解两方面进行优化。

3 编辑距离下的同源序列高效检索算法

因为无法避免同源序列两两比对的步骤，需要实现时间复杂度的降维只能从缩小需要比对的序列的规模入手，找寻一种低相似度序列的快速过滤算法。

3.1 编辑相似度上界E_s的确定

3.2.1 最长公共子序列问题

给定一个序列X = ⟨x_1, x_2, … , x_m⟩，另一个序列Z = ⟨z_1, z_2, … , z_k⟩满足如下条件时称为 X 的子序列 (subsequence) ，即存在一个严格递增的的 X 的下标序列⟨i_1, i_2, … , i_k⟩，对所有j=1, 2, ..., k，满足$x_{i_j} = z_j$。例如，Z = ⟨B, C, D, B⟩是X = ⟨A, B, C, B, D, A, B⟩的子序列，对应的下标序列为⟨2，3，5，7⟩。

给定两个序列 X 和 Y，如果 Z 既是 X 的子序列，也是 Y 的子序列，我们称它是 X 和 Y 的公共子序列 (common subsequence) 。例如，如果X = ⟨A, B, C, B, D, A, B⟩，Y = ⟨B, D, C, A, B, A⟩，那么序列⟨B, C, A⟩就是 X 和 Y 的公共子序列，但并非最长公共子序列 (longest common subsequence, LCS) 。因为⟨B, C, B, A⟩也是 X 和 Y 的公共子序列，但其长度为 4，大于⟨B, C, A⟩的长度。⟨B, C, B, A⟩是 X 和 Y 的最长公共子序列，⟨B, D, A, B⟩也是，因为 X 和 Y 不存在长度大于等于 5 的公共子序列。[20]

基于 LCS 也可以定义两个序列的相似度new_s。LCS 的长度越长，则两个序列的相似度越大。

LCS 从某种意义上来说，可以看作增删代价取 0，改操作换成相同奖励的 Levenshtein 距离。因为求编辑距离过程中，两个序列不需要进行编辑的对应字符一定包含于它们的 LCS 中，所以两个序列的 LCS 长度一定大于编辑距离下两个序列重合的字符数目。因此可以得到

$$\begin{aligned} E_s(a,b) &= 1 - E_d(a,b) \\ &= \frac{\max(|a|,\ |b|) - lev(a,\ b)}{\max(|a|,\ |b|)} \\ &\le \frac{|LCS(a,b)|}{\max(|a|,|b|)} = new_s \end{aligned} \tag{3.1}$$

求解序列对的编辑距离本质上是离散上的最优化问题。Levenshtein 距离的计算对全域进行检索，使得算法可以跳出极值点，收敛到全局最优解。而基于 LCS 的编辑距离计算，由于 LCS 限制了最优化问题的求解域，求解函数只能在一定范围内迭代，逼近局部

最优解，即极小值。

而计算两个序列的 LCS 长度，仍是基于动态规划思想。具体的转移方程可以表述为

$$\mathrm{LCS}(X_i, Y_j) = \begin{cases} \emptyset & if\ \ i = 0\ \ or\ \ j = 0 \\ \mathrm{LCS}(X_{i-1}, Y_{j-1})\hat{}x_i & if\ \ i,j > 0\ \ and\ \ x_i = y_j \\ \max\left(LCS(X_{i-1}, Y_j), LCS(X_i, Y_{j-1})\right) & if\ \ i,j > 0\ \ and\ \ x_i \neq y_j \end{cases} \tag{3.2}$$

而这种方法虽然可以得到编辑相似度的上界，却并没有缩减时间复杂度，仍需要$O(mn)$的时间复杂度。因此我们还需要探索新的上界表示方法。

3.2.2 Jaccard 相似度

Jaccard 相似度 (Jaccard index) 是用于衡量数据集的相似性、相异性和距离的统计量。测量两个数据集之间的 Jaccard 相似度是将所有数据集共有的特征数除以属性数的结果[21]。

$$\mathrm{J}_s(\mathrm{A}, \mathrm{B}) = \frac{|A \cap B|}{|A \cup B|} \tag{3.3}$$

由于计算序列相似度需要考虑同一个元素出现的次数，我们采用多重集合 (multi-sets 或 weighted sets) 来代替集合。具体来说，多重集合 A 被定义为一个索引函数χ_A：当集合元素对应的全集$\mathcal{U} \to \mathbb{N}$，A 中的元素 x 的出现次数（出现次数为 0 的元素不在集合中记录）。两个多重集合的交集的索引函数是函数的最小值，而对于并集来说，则是最大值。所以我们引入加权 Jaccard 相似度的概念，见式 3.4。

$$\mathrm{J}_s(\mathrm{A}, \mathrm{B}) = \frac{\sum_{x \in \mathcal{U}} \min(\chi_A(x), \chi_B(x))}{\sum_{x \in \mathcal{U}} \max(\chi_A(x), \chi_B(x))} \tag{3.4}$$

Jaccard 相似度是一种索引函数只取值{0,1}的特殊加权 Jaccard 相似度。为了方便描述，在之后的文章中，Jaccard 相似度特指加权 Jaccard 相似度。

Jaccard 相似度是编辑相似度的上界。证明如下：

存在序列 a 和 b，集合 A、B 分别与之对应。LCS 是一种包含集合相对位置信息的交集，而$A \cap B$不要求位置信息，重合率会更高。因此$|LCS| \leq |A \cap B|$。例如，序列 a=ABC，序列 b=ACB，a 和 b 的 LCS 长度只能是 2，但是交集大小是 3。因为 BC 的倒置会影响 LCS 中的基于位置关系的转移方程，但并不影响集合元素数量关系。

假设$|a| \geq |b|$，则$\max(|a|, |b|) = |a|$。$|A \cup B| = (|A| + |B| - |A \cap B|) \geq |A| = |a|$。所以有

$$\frac{|A \cap B|}{|A \cup B|} \geq \frac{|LCS(a,b)|}{\max(|a|,|b|)} \geq E_s(a,b) \tag{3.5}$$

利用 Jaccard 相似度可以实现低编辑相似度序列的快速过滤。对于序列S_1, S_2，存在两个常数θ和α $(\theta, \alpha \epsilon [0,1],\ \theta \geq \alpha)$ ，只有当 Jaccard 相似度$J_s(S_1, S_2) < \alpha$，才有编辑相似度$E_s(S_1, S_2) < \theta$的情况存在。我们考虑它的逆否命题：若$E_s(S_1, S_2) < \theta$的情况不存在，即任意$E_s(S_1, S_2) \geq \theta$，则由式 3.5 可推导得$J_s(S_1, S_2) \geq E_s(S_1, S_2) \geq \theta$。因为$\theta \geq \alpha$，所以$J_s(S_1, S_2) \geq \alpha$。从而原命题成立。

因此使用阈值为α的$J_s(S_1, S_2)$对序列进行过滤时，虽然仍存在假阳性 (false positive, FP) 的情况，过滤后的序列中依然有$E_s(S_1, S_2) < \theta$的对；但是假阴性 (false negative, FN) 可以被完全消除，即所有$E_s(S_1, S_2) \geq \theta$的序列对都被保留。

3.2 快速过滤算法的时间复杂度分析和优化

Jaccard 相似度作为编辑相似度的上界，用来实现低编辑相似度序列的快速过滤。$O(l^2mn)$的朴素算法，可以采用$O(l^2(m+n))$的过滤算法来大大缩减l的值域，最后将时间复杂度降为$O(l'^2mn)(l' \ll l)$。

然而，快速过滤算法$O(l^2(m+n))$的时间复杂度仍是不可接受的。$O(l^2)$来源于序列的两两比对，$O(m+n)$来源于使用 hash 来计算集合之间的交并集。因此，找到一种聚类算法来对l进行降维，并且运用兼顾效率与准确性的 hash 算法，是提升快速过滤算法速度的关键。

3.3 Order Min Hash 模型介绍及复现

3.3.1 术语介绍

（1）k-mer

在生物信息学中，k-mers 是包含在生物序列中的长度为 k 的子串。所以，一条生物序列就是很多个不同的 k-mer 的集合。

例如，序列为 AHFUWFU。令 k=2，那么这条序列中所有的 2-mer 组成的集合为{AH, HF, FU, UW, WE}。需要注意的是，FU 在序列中出现了两次，但是在集合中只能出现一次。这是因为集合中不能包含相同的元素，但元素的权值会进行相应的增加。

尽管用 k-mer 的方式来表示每条序列，然后再通过判断每条序列中 k-mer 集合的重叠率，就可以得出序列的相似度。但是，一条序列得到的 mer 集合的元素个数并不少。换而言之，长度为 L 的序列有 L-k+1 的 k-mers。k 的取值影响相似度度量的准确性以及内存大小。k 取值过小：特别是当 k=1 时，k-mer 退化为单字符，不携带任何序列位置

信息，大大降低了相似度的可靠性。k 取值过大：假定 k=5，那么每个 mer 中就会包含 5 个字符，序列需要的内存空间大概是原序列大小的 5 倍。因为原序列中的每个字符（除去开头和结尾的字符有特殊性）都会出现在 5 个 mer 中。以 k-mer 的方式来存储序列会消耗大量的内存。

因此，我们需要把上面的 k-mer 集合替换成规模小很多的散列值集合。这样，通过比较两个序列的散列值集合的相似度，就可以估计 k-mer 的相似度。散列值会损失部分精度，因此这样得到的相似度只是原来相似度的近似值。

（2）哈希函数 Hash function

哈希函数是可将任意大小的数据映射到固定大小值的一类函数。哈希函数及其相关的数据结构——散列表 (hash table) 用于数据存储和检索，以便每次检索在较小且几乎恒定的时间内访问数据。

Hash 需要的存储空间量仅略大于数据本身所需的总空间。当实际存储的关键字数目比全部的可能关键字总数要小时，采用散列表就成为直接数组寻址的一种有效替代。因为散列表使用一个长度与实际存储的关键字数目成反比的数组来存储。散列是一种时空复杂度高效的数据访问形式，避免了列表和结构化树的非恒定访问时间，以及直接访问的指数级存储要求。

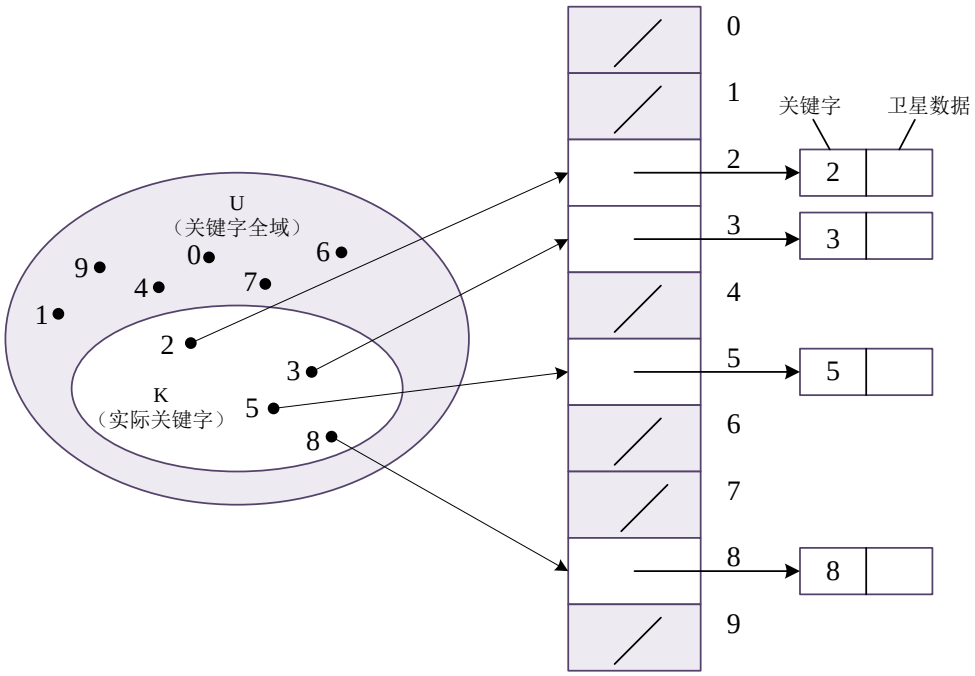

图 3.1　直接寻址表[20]

当关键字的全域 U 比较小时，直接寻址是一种简单而有效的技术。如图 3.1 所示，

假设动态集合中每个元素都取自于全域 U={0, 1, …, m-1}中的一个关键字，且集合中任意两个元素都不具有相同的关键字。我们用直接寻址表 (direct-address table) 表示动态集合，记为 T[0..M-1]。其中每个位置都称为一个槽 (slot) ，对应全域 U 中的一个关键字。槽 k 指向集合中一个关键字为 k 的元素。T[k]=NIL 代表该集合中没有关键词为 k 的元素。

直接寻址技术具有如下缺点：

(a) 如果全域 U 很大，空间复杂度O(U)将不可接受，超出标准计算机内存；

(b) 如果实际需要存储的关键字集合|K| ≪ |U|，分配给 U 的大部分空间实际都是 NIL 的，造成空间浪费。

当|K| ≪ |U|时，散列表需要的存储空间远小于直接寻址表。散列表的空间复杂度可以降至O(|K|)，同时查找一个元素的效率仍能得到保持，只需要O(1)的时间。

在直接寻址方式下，具有关键字 k 的元素被存放在槽 k 中。在散列方式下，该元素存放在槽 h(k)中，即利用散列函数 h，由关键字 k 计算出槽的位置。函数 h 将关键字的全域 U 映射到散列表 T[0..m-1]的槽位上：

$$h: U \to \{0, 1, \ldots, m-1\} \quad (m \ll |U|) \tag{3.6}$$

图 3.2 描述了这种映射关系。我们可以说一个具有关键词 k 的元素被散列到槽 h(k)上，也可以说 h(k)是关键词 k 的散列值。散列函数大大降低了对存储空间的需求。

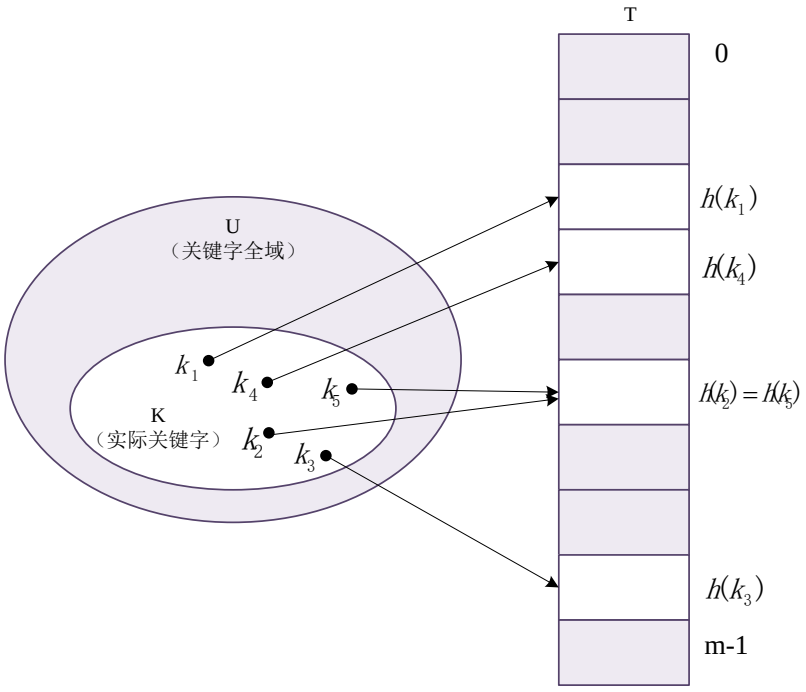

图 3.2 散列表[20]

然而，散列会产生一个新的问题：两个不同的关键字有可能映射至同一个槽中。这种情况也被称为冲突 (collision) 。

理想的解决方案是找到一种可行且有效的方法规避所有冲突。但由于$|U| > m$，所以至少存在两个关键字，它们的散列值相同，被划分在同一个槽中。因此完全避免冲突是无法实现的。一方面，我们可以通过精心设计散列函数来尽量降低冲突发生概率；另一方面，当冲突不可避免发生时，我们仍需要一种解决冲突的办法。

（3）全域散列 universal hashing

如果针对某个特定的散列函数去选择散列关键字，增加冲突可能性，是非常容易实现的。事实上可以将 n 个关键字全部散列在同一个槽中，平均检索时间就从常数级别O(1)变为了线性O(n)。任意一个特定的散列函数都存在如上所述的致命缺点。

唯一有效的改进方法就是采用全域散列 (universal hashing) ：随机选择散列函数，使之独立于要存储的关键字。不论被给予什么样的关键词集合，全域散列都能保证其平均性能。

设$\mathcal{H}$是一组有限散列函数，它将给定的关键词全域 U 映射到{0, 1, …, m-1}中。一个函数组如果满足：对每一对不同的关键字k, l ∈ U，h(k) = h(l)的散列函数h ∈ $\mathcal{H}$的个数至多为$|\mathcal{H}|$/m，就被称为全域的 (universal)。要而言之，如果从$\mathcal{H}$中随机选择一个散列函数，当关键字k ≠ l时，两者发生冲突的概率小于等于 1/m，恰好就是从集合{0, 1, …, m-1}中独立地随机选择 h(k)和 h(l)时发生冲突的概率。

我们日常使用的全域散列函数类构造非常简单。首先需要选择一个大素数 p (p≥m) ，使得每一个可能的关键字 k 都落在[0, p-1]内。设Z_p表示集合{0, 1, …, p-1}，Z_p^*表示集合{1, 2, …, p-1}。

对于任何a ∈ Z_p^*和任何b ∈ Z_p，定义散列函数h_{ab}。利用一次线性变换，再进行模 p 和模 m 的归约，有

$$h_{ab}(k) = \big((ak + b)\ mod\ p\big)\ mod\ m \tag{3.7}$$

例如，当 p=17，m=6 时，$h_{3,4}(8) = 5$。

依据生日悖论 (Birthday problem) ，假设存在 n 个人，他们的生日是[1,365]中的随机整数。当n > 365时，一定存在两个人生日相同；当n ≤ 365时，n 个人的生日都不相同的概率$p_n = \frac{P_{365}^n}{365^n}$。当 n = 23 时，上述结果约为 0.46，即有超过 50% 的概率有人生日相同。而在哈希构造中，可以解释为当检验次数超过$\sqrt{p}$，就会有较大概率发生错误。因此

要求 p 足够大。

考虑 p 是合数的情况。假设$N = g * n$，$M = g * m$，N 和 M 存在最大公因数 g。$N \bmod M = r$可以转换成$N = Mq + r$，即$gn = gmq + r$。其中 q 是商，r 是余数。看起来 r 的取值范围仍是{0, 1, 2, …, M-1}。但$gn = gmq + r$仍可以继续转换为$n = mq + r/g$。因为 n 和 mq 都是整数，所以 r 一定能整除 g。而 r/g 的取值范围{0, 1, 2, …, m}={0, 1, 2, …, M/g}，因此 r 的实际取值范围是{0, k, 2*k, 3*k, …, m*k}，缩小了 k 倍。因此还要求 p 是个足够大的质数。

所有这样的散列函数构成的函数族为

$$\mathcal{H}_{pm} = \{h_{ab}: a \in Z_p^*, b \in Z_p\} \tag{3.8}$$

每一个散列函数h_{ab}都将Z_p映射到Z_m。这一类散列函数具有一个良好的性质，即输出范围的大小 m 是任意的，不局限于大质数。由于对 a 来说有 p-1 种选择，对 b 来说有 p 种选择，所以$\mathcal{H}_{pm}$中包含 p(p-1)个散列函数。

以下给出散列函数族$\mathcal{H}_{pm}$是全域的证明。

考虑Z_p中的两个不同关键字 k 和 l，即$k \neq l$。对于某一个给定的散列函数h_{ab}，设

$$r = (ak + b) \bmod p \tag{3.9}$$

$$s = (al + b) \bmod p \tag{3.10}$$

因为$r - s \equiv a(k - l) \pmod p$，而根据已知$a > 0$，$k - l \neq 0$，$a(k - l) \neq 0$。p 为质数，与合数$a(k - l)$互质，因此$a(k - l) \pmod p \neq 0$，得$r \neq s$。在模 p 层次上，计算任何$h_{ab} \in \mathcal{H}_{pm}$，不同的输入 k 和 l 一定会被映射到不同的值 r 和 s，不会产生冲突。而给定 r 和 s 后，也可以解出 a 和 b 的表达式：

$$a = \left((r - s)((k - l)^{-1} \bmod p)\right) \bmod p \tag{3.11}$$

$$b = (r - ak) \bmod p \tag{3.12}$$

其中$(k - l)^{-1}$是$(k - l)$在模 p 意义下的逆元。因为除法取余运算中不具有分配律，需要把除法运算转换为乘法计算，即乘逆元取余。因此，在数对$(a, b)(a \neq 0)$与数对$(r, s)(r \neq s)$之间存在着一一对应的关系。所以，对于任意给定的输入对 k 和 l，如果从$Z_p^* \times Z_p$中均匀地随机选择(a, b)，则结果数对(r, s)在模 p 意义下，就等可能地为任何不同的数对值。

综上所述，当 r 和 s 为模 p 下随机选择的不同的值时，不同的关键字 k 和 l 发生冲突的概率等于$\Pr\{r \equiv s \pmod m\}$。对于某个给定的 r 值，s 的可能取值空间为 p-1，其中

满足$s \neq r$且$s \equiv r(\mathrm{mod}\ m)$的 s 值的数目至多为：

$$\lceil p/m \rceil - 1 \leq \left(\frac{p+m-1}{m}\right) - 1 = (p-1)/m \tag{3.13}$$

因此，s 与 r 发生冲突的概率至多为$((p-1)/m)/(p-1) = 1/m$。所以，对于任何不同的数对$k, l \in Z_p$，有

$$\Pr\{h_{ab}(k) = h_{ab}(l)\} \leq 1/m \tag{3.14}$$

得以证明$\mathcal{H}_{pm}$是全域的。

（4）最小哈希 MinHash

MinHash 是一种快速估计两个集合的相似程度，即 Jaccard 相似度$J_s(S_1, S_2)$，而无需直接计算集合之间交集和并集的技术。

给定一个全集$\mathcal{U}$和它的子集$\mathcal{S} \subseteq \mathcal{U}$，MinHash 定义如下：假设存在包含 D 个散列函数（或随机排列）的函数族 $\{\hbar_d\}_{d=1}^{D}$，$\hbar(x)$ 表示一个将$\mathcal{U}$上的关键字x映射到$\mathcal{S}$上的散列函数。$\hbar_{min}(\mathcal{U}) = y$ 被定义为如果 $y \in \mathcal{U}$，且与$\mathcal{U}$的所有元素中比较 $\hbar(y)$ 具有最小散列值，这也被称作$\mathcal{S}$的一个 MinHash。

假设集合$S = \{s_1, s_2, s_5, s_7, s_{10}\}$被散列成$\{4, 6, 2, 1, 8\}$，那么$\hbar_{min}(S) = s_7$。

将$\hbar_{min}$应用到集合 A 和 B 上，假设不存在哈希冲突，$\hbar_{min}(A) = \hbar_{min}(B)$当且仅当在$A \cup B$的所有元素中，具有最小哈希值的元素在$A \cap B$中。这种情况发生的概率恰好等于 Jaccard 相似度。因此，

$$\Pr[\hbar_{min}(A) = \hbar_{min}(B)] = J_s(A, B) \tag{3.15}$$

一个散列函数很容易受到特殊数据的影响。为了获得无偏相似度估计，我们进行多个独立的随机排列：

$$\widehat{J_s}(A, B) = \frac{\sum_{d=1}^{D} \mathrm{if}(\hbar_{dmin}(A) = \hbar_{dmin}(B))}{D} \tag{3.16}$$

如果 state 为真，$\mathrm{if(state)} = 1$；否则$\mathrm{if(state)} = 0$。当$D \to \infty$的时候，$\widehat{J_s}(A, B) \to J_s(A, B)$。

而由（3）分析可知，哈希冲突无法避免，但如果我们在 MinHash 中使用全域散列函数族，可以最大程度减少这种冲突，使得冲突概率不高于$1/|\mathcal{S}|$。因此，使用 MinHash 来衡量 Jaccard 相似度只是一种近似估计，其准确度受到 k-mer 的参数选择以及散列函数族的选择等影响。

假设我们在函数族中使用了 k 个随机选择的哈希函数，满足$\hbar_{min}(A) = \hbar_{min}(B)$的函数个数为 y。$y/k$可以视为$J_s(A, B)$的无偏估计量。根据切诺夫界 (Chernoff bound)，误差期望$E(|y/k - J_s(A, B)|) = O(1/\sqrt{k})$。因此如果 k=400 时，预期误差仅为 0.05。

例如，存在$S_1 = \{a, d, e\}$，$S_2 = \{c, e\}$，全集$\mathcal{U} = \{\mathrm{a, b, c, d, e}\}$。集合可以表示为表 3.1 的形式。

表 3.1　集合表示

行号	元素	S_1	S_2	类别
1	a	1	0	Y
2	b	0	0	Z
3	c	0	1	Y
4	d	1	0	Y
5	e	1	1	X

其中，列表示集合，行表示相应的元素，对应的值为 1 表示某个集合中有某个值，0 则代表没有。S_1和S_2的每一行元素可以按取值是否相等，分为以下三类：

(a) X 类，两者值均为 1。如表 3.1 中第 5 行，两个集合都具有元素 e；

(b) Y 类，两者取值不相同，一个为 0，另一个为 1；如表 3.1 中第 1 行，表示S_1中具有元素 a，而S_2中没有；

(c) Z 类，两者值均为 0。如表 3.1 中第 2 行，表示S_1和S_2中都没有元素 b。

设哈希函数$h_1 = (i + 1)\%5$，其中 i 代表行号。h_1作用于集合S_1和S_2，可以得到表 3.2 的结果表示。

表 3.2　h_1下的集合表示

行号	元素	S_1	S_2	类别
1	e	1	1	X
2	a	1	0	Y
3	b	0	0	Z
4	c	0	1	Y
5	d	1	0	Y
MinHash		e	e	

此时，$h_{min}(S_1)$=e，$h_{min}(S_2)$=e，即S_1和S_2的 MinHash 值都是 e。

所有标记为 Z 类的行都可以被忽略，因为 Z 对应的元素完全可以从全集$\mathcal{U}$中删除，对两个集合的相似度计算不产生任何影响。我们讨论在哈希函数h_1均匀分布的情况下，由于h_1将原始行号均匀分布到新的行号，那么就可以认为在新行号排列下，任意一行出现 X 类的情况的概率为$|\mathrm{X}|/(|\mathrm{X}| + |\mathrm{Y}|)$。所以，第一个出现 X 类行的行出现 X 类的概率也为$\mathrm{P} = |\mathrm{X}|/(|\mathrm{X}| + |\mathrm{Y}|) = \mathrm{J}_s(\mathrm{X}, \mathrm{Y})$。

继续假设存在哈希函数$h_2 = (i - 1)\%5$，那么可以得到表 3.3。

表 3.3 $\hbar_2$下的集合表示

行号	元素	S_1	S_2	类别
1	b	0	0	Z
2	c	0	1	Y
3	d	1	0	Y
4	e	1	1	X
5	a	1	0	Y
MinHash		d	c	

哈希函数族中存在 h 个散列函数 $(\mathrm{h} \ll \mathcal{U})$，因此我们需要为每个集合计算 h 次 Minhash 值，用这 h 个 Minhash 值组成一个摘要，来表示当前集合，如表 3.4 所示。

表 3.4 $\hbar_1$和$\hbar_2$下的MinHash摘要

哈希函数	S_1	S_2
$\hbar_1 = (i+1)\%5$	e	e
$\hbar_2 = (i-1)\%5$	d	c

令 M 表示 Minhash 摘要中集合对应行相等的次数，即函数族中满足两个集合最小散列值相等的函数个数。表 3.4 中，M=1，因为在哈希函数$\hbar_1$散列下，S_1和S_2的 MinHash 值相同，而在$\hbar_2$下则不同。那么可以发现，

$$\mathrm{M} \sim \mathrm{B}(\mathrm{h}, \mathrm{J}_s(S_1, S_2)) \tag{3.17}$$

M 符合次数为 h，概率为$\mathrm{J}_s(S_1, S_2)$的二项分布，而期望$\mathrm{E}(\mathrm{M}) = \mathrm{h} * \mathrm{J}_s(S_1, S_2) = 2 * (1/4) = 0.5$。也就是说，每运用 2 个散列函数计算 Minhash 摘要时，可以期望有 0.5 个元素对应相等。综上所述，MinHash 在对原集合进行散列的基础上，保证了集合的相似度不受破坏。

（5）局部敏感哈希 Locality-Sensitive Hashing

局部敏感哈希 （Locality-Sensitive Hashing, LSH） 是一种可以以较高概率将相似的输入项散列到相同的槽中的算法技术。

与传统的哈希算法理念不同，LSH 将产生 Hash 冲突的概率最大化，而并非极力避免这种冲突。LSH 能使 2 个相似度很高的数据以较高的概率映射成同一个哈希值，而 2 个相似度很低的数据以极低的概率映射成同一个哈希值的特性。因此 LSH 能高效处理海量高维数据的聚类和最近邻搜索问题。

我们提出一种问题场景。存在长度为 d 的两个碱基序列S_1和S_2，$\mathrm{r} < \mathrm{d}$，如果S_1和S_2之间相差不超过 r 个字符，就称S_1和S_2相似。给定一个序列 $\mathrm{G}\ (|\mathrm{G}| = \mathrm{n} \gg \mathrm{d})$，对于每一

个询问的长度为 d 的序列，在 G 中找出相似的子串。

如果我们采用直接比较的方式，时间复杂度为O(nd)。我们需要找到一种时间复杂度与 d 相关（关于 d 的多项式）而与 n 无关的算法。

在预处理阶段，将 G 中所有长度为 d 的子串通过随机选择的局部敏感哈希 (LSH) $h_1, h_2, \dots, h_l$分别散列到哈希表$A_1, A_2, \dots, A_l$的集合中。对于每一个长度为 d 的询问串 q，查找A_1中的$h_1(q)$，A_2中的$h_2(q)$，…，A_l中的$h_\mathrm{l}(q)$，令并集$\mathrm{Q} = A_1[h_1(q)] \cup A_2[h_2(q)] \cup \dots \cup A_\mathrm{l}[h_\mathrm{l}(q)]$。那么最后只需要将 Q 中的每一项与 q 进行相似度计算，就能完成对 n 的降阶。

我们定义一种局部敏感哈希函数h_i，如果对于一个与询问串 q 相似的串 g，h_i存在 p 的概率返回 g。

$$\Pr\big(\mathrm{g} \in h_\mathrm{i}(q)\big) = p \tag{3.18}$$

$$\Pr\big(h_\mathrm{i}(q) = h_\mathrm{i}(g)\big) = p \tag{3.19}$$

由于存在哈希函数族$h_1, h_2, \dots, h_l$，因此 g 被找到的概率为

$$\begin{aligned}\Pr(\mathrm{g} \in \mathrm{Q}) &= 1 - \Pr(\mathrm{g} \notin \mathrm{Q}) \\ &= 1 - \textstyle\sum_{i=1}^{l} \Pr\big(\mathrm{g} \notin h_\mathrm{i}(q)\big) = 1 - (1-p)^l\end{aligned} \tag{3.20}$$

假设 p=0.1，我们能得到如下关系表 3.5。

表 3.5　p=0.1时，l与函数结果的关系

l (p=0.1)	$1-(1-p)^l$
1	0.1
2	0.19
4	0.344
8	0.570
16	0.815
32	0.966

分析可知，随着l的小幅度增大，g 被找到的概率会无限趋近 1。

当$\mathrm{n} = |\mathrm{G}|$的时候，算法的空间复杂度显然为$\mathrm{O}(l\mathrm{n})$。在预处理阶段，需要对每一个子串计算一遍散列值，因此时间复杂度为$\mathrm{O}(l\mathrm{n})$；而在查找阶段，运用 MinHash 估计聚类后数据的 Jaccard 相似度，时间复杂度为$\mathrm{O}(l\mathrm{d} + |\mathrm{Q}|\mathrm{d})$。

对于解决前文提出的问题，我们可以设计这样一个 LSH 函数，对于 G 中每个长度为 d 的子串 s，从$\{1 \dots \mathrm{d}\}$均匀随机选择k $(k < d)$ 个置换$i_1, i_2, \dots, i_k$。定义 LSH 函数$\mathrm{H} = \{\mathrm{h}: \sum^d \to \sum^k\}$，其中$\sum = \{a, c, g, t\}$，那么

$$\mathrm{h(s)} =< \mathrm{s}[i_1], \mathrm{s}[i_2], \dots, \mathrm{s}[i_k] > \tag{3.21}$$

例如，当 k=2，$i_1 = 2$，$i_2 = 5$时，h(“acgtacgt”) = “ca”。

每一位 i 在集合中都有 d 种选择，因此总共有d^k个不同的置换函数 h。

当$\mathrm{i} = 1 \dots l$，随机选择一个 LSH 函数h_1，对于每个符合条件的子串的起始位置 v，将 $(h_i(\mathrm{G[v \dots v + d - 1]}), \mathrm{v})$ 放入散列表A_i中。为了方便接下来的多组询问，在预处理阶段我们将每个散列表A_i中的元素按照h_i值的大小进行排序。因此这部分总时间复杂度近似 $\mathrm{O}(lnd \log n)$ 。在询问阶段，对于$\mathrm{i} = 1 \dots l$，计算出$h_i(q)$，在每个A_i中二分查找与$h_i(q)$相同的值对应的位置信息 v，最后只需要比较序列 q 和$\mathrm{G[v \dots v + d - 1]}$的相似度。时间复杂度为 $\mathrm{O}(ld \log n)$ 。

因为函数 h 具有以较高的概率将相似的序列散列到同一个槽中的性质。 如果序列s_1和s_2相似（即序列间差异的位数小于 r），有

$$\Pr\big(h(s_1) = h(s_2)\big) \geq (1 - \frac{r}{d})^k \tag{3.22}$$

假阴性情况（本来相似的序列对在所有散列表中都不在一个槽中）发生的概率f_n为

$$f_n \leq [1 - (1 - \frac{r}{d})^k]^l \tag{3.23}$$

对于阈值ρ_{f_n}，有$f_n \leq \rho_{f_n}$。因此敏感度就可以定义为$1 - \rho_{f_n}$，这通常是提前设置的。也就是说，如果要求 95%的相似序列对被找到，ρ_{f_n}就需要调整为 0.05。

对于一个固定的值l，我们需要找到一个 k 满足

$$f_n \leq [1 - (1 - \frac{r}{d})^k]^l \leq 1 - \rho_{f_n} \tag{3.24}$$

设每个散列函数的假阳性概率为f_p。LSH 函数会返回一组包括真解和假阳性解的结果。因为需要对所有假阳性结果进行检验来筛除，f_p对运行时间的影响为$\mathrm{O}\big(lnf_p\big)$。

假设一个随机生成的长度为 d 的 DNA 序列 s 满足$f_a = f_c = f_g = f_t = 0.25$的频率分布，那么有$f_p = \Pr\big(h(s) = h(\mathrm{q})\big) = 0.25^k$。所以$\mathrm{O}\big(lnf_p\big) = \mathrm{O}(ln0.25^k)$，因此我们需要尽可能地选择较大的参数 k。若 d=100，r=20，将l视为固定的值，求解 k，有

$$[1 - (1 - \frac{20}{100})^k]^l = (1 - 0.8^k)^l \leq 0.05 \tag{3.25}$$

$$1 - e^{\frac{\ln 0.05}{l}} \leq 0.8^k \tag{3.26}$$

$$k \leq \frac{\ln(1 - e^{\frac{\ln 0.05}{l}})}{\ln 0.8} \tag{3.27}$$

当l取不同的数值时，对应的更为直观的结果可见表 3.6。

表 3.6 l, k与lf_p的对应关系

l	k ≤	$l \times 0.25^k$
20	8.8	9.53E-05
30	10.5	1.34E-05
40	11.8	3.23E-06
50	12.7	1.06E-06
60	13.5	4.21E-07
80	14.8	9.77E-08
100	15.8	3.13E-08

虽然使用 LSH 不能保证找到解决方案，但找到解决方案的概率较高。这个概率可以通过增加 LSH 函数的数量来提升。

（4）中提出的 MinHash 方法，仍然需要遍历所有的集合对，才能挖掘所有相似的集合对，无法对 $O(n^2)$ 的复杂度进行优化。接下来将以 LSH 方法解决集合间两两比对的难题进行举例说明。

现在有 5 个集合，对应的 Minhash 摘要见表 3.7。

表 3.7 集合的Minhash摘要

	S_1	S_2	S_3	S_4	S_5
区间1	b	b	a	b	a
	c	c	a	c	b
	d	b	a	d	c
区间2	a	e	b	e	d
	b	d	c	f	e
	e	a	d	g	a
区间3	d	c	a	h	b
	a	a	b	b	a
	d	e	a	b	e
区间4	d	a	a	c	b
	b	a	c	b	a
	d	e	a	b	e

如上的集合摘要采用了 12 个不同的哈希函数散列，之后将结果分成了 B = 4 个区间。根据（4）中的分析，任意两个集合 (S_1, S_2) 对应的 Minhash 值相等的概率 r = $J_s(S_1, S_2)$。在区间 1 中，$\Pr(S_1 = S_2) = r^3$。因此如果$J_s(S_1, S_2)$足够大，S_1和S_2在区间 1 内的三个最小哈希值就可能完全一致，S_1和S_2可以被认定为相同。也就是说，$\Pr(S_1 \neq S_2) = 1 - r^3$。

现在有 4 个区间，其他区间与第一个区间等价，所以有$\Pr(S_1 \neq S_2) = (1 - r^3)^4$。那么至少存在一个区间上$S_1 = S_2$的概率$\Pr(S_1 = S_2) = 1 - (1 - r^3)^4$。函数图像如下。

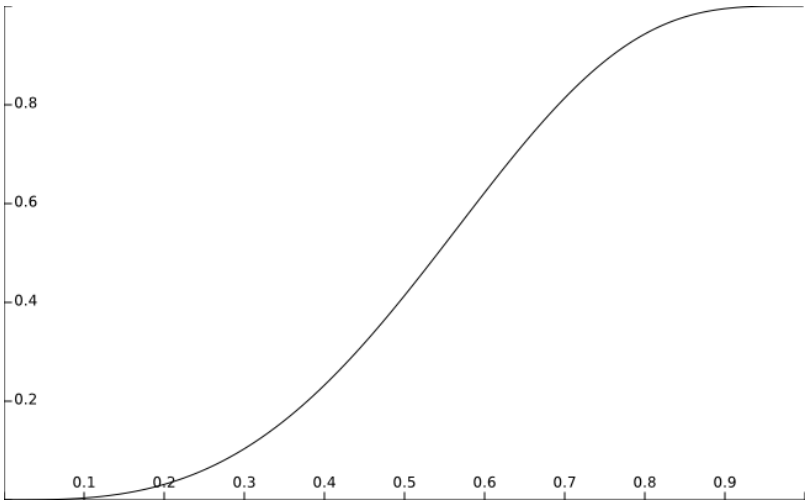

图 3.3　4个区间，12个哈希函数下的冲突概率函数

如果令总区间个数为 B，每个区间内的行数为 C，那么上面的公式可以表示为 Pr（B 个区间中至少有一个区间中两个集合相等）=$1 - (1 - r^C)^B$。

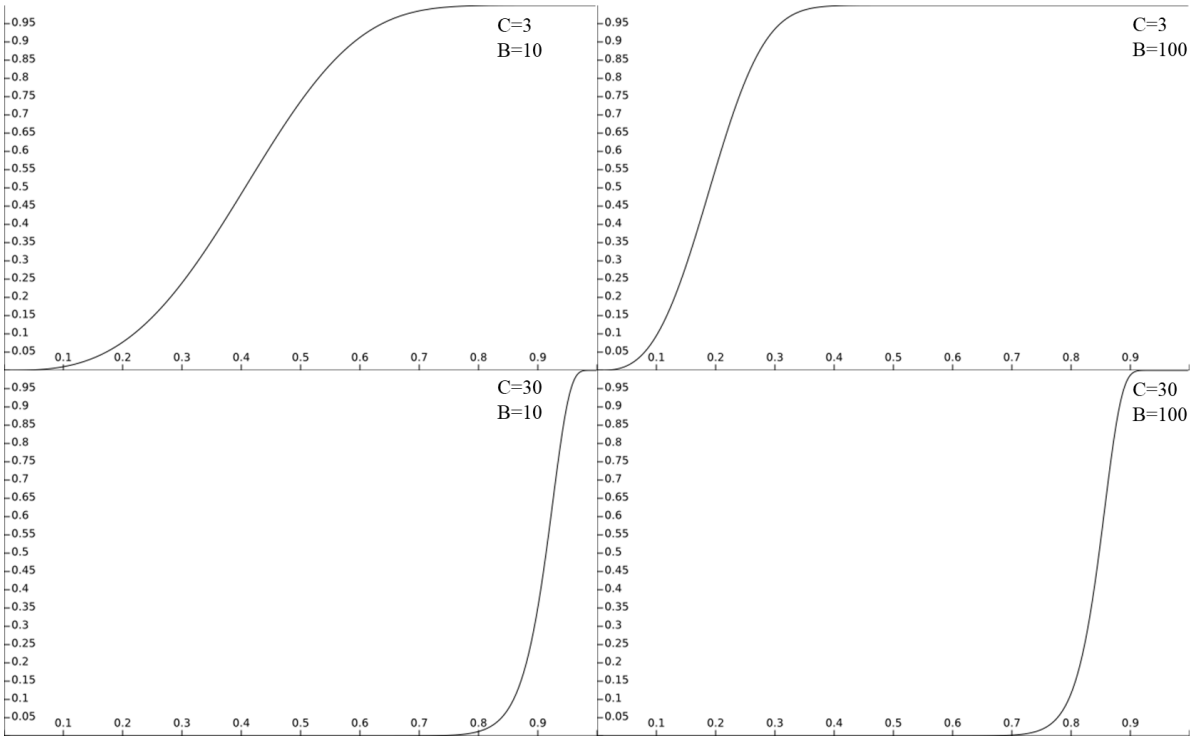

图 3.4　不同B和C的值对冲突概率函数的影响

令 r = 0.4，C=3，B = 100。上述公式计算的概率为 0.9986585。这表明两个 Jaccard 相似度为 0.4 的集合在至少一个区间内冲撞的概率达到了 99.9%。因此我们只需要选取

合适的 B 和 C，和一个冲撞率很低的 hash 函数，就可以使相似的集合至少在一个区间内冲撞，这样也就达成了将相似的集合聚类的目的。因为 B 和 C 都是常量，所以 LSH 的时间复杂度是线性O(n)的。由于聚到一起的集合相比于整体较少，所以在这小范围内互相比较的时间开销也可以计算为常量，那么总体的计算时间也是O(n)。

在上文基础上，我们引入离散空间上的广义局部敏感哈希 (Generalized Locality Sensitive Hashing) 的定义。假设有一个在 d 维空间$\mathbb{R}^d$上，包含 n 个数据点$p = (p_1, \dots, p_d)$的集合 P。对于任意两点 p 和 q，它们之间的距离定义为

$$||p-q||_s = \left(\sum_{i=1}^{d}|p_i - q_i|^s\right)^{\frac{1}{s}} \tag{3.28}$$

对于任意的 S>0，这个距离函数被称为标准化l_s。如果$||p-q||_s \le R$，则称 p 是 q 的 R 近邻 (R-near neighbor) 。如图 3.5，p_1，p_2，p_3都是 q 的 R 近邻，p_4则不是。

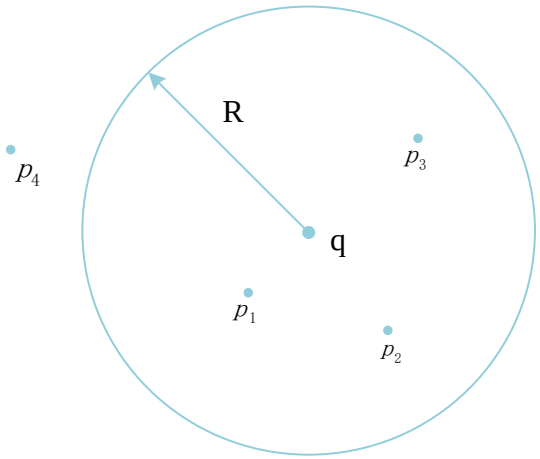

图 3.5　离散空间上的R近邻

从几何学的角度理解 LSH，本质就是一种投影。首先根据散列函数进行散列操作，这个函数在二维空间上可以表示为一条直线，使得空间上相邻的点投影在这条直线的同一段区间里，即散列到同一个桶中。这种 LSH 和连续函数 mod k 的散列方式是有本质区别的，无法进行精准的归类，让所有相邻的点都被散列在一个桶中，而不相邻的点也无法保证一定不在同一个桶中。但是 LSH 在一定程度上可以区分查询点的相近点和较远点。

上述最近邻 (near neighbor, NN) 问题可以扩展到近似最近邻(Approximate near neighbor, ANN) 问题[22] ，也被称为 Randomized c-approximate R-near neighbor ((c,R)-NN)。

给定点集P $\subseteq \mathbb{R}^d$，查询点q $\in \mathbb{R}^d$，查询范围 R>0 和近似因子 c>1，(c, r)-NN 问题的输出如下：若存在$p \in$ P，满足$||p-q||_s \le$ R，则输出某点$p' \in$ P，满足$||p'-q||_s \le$ cR。

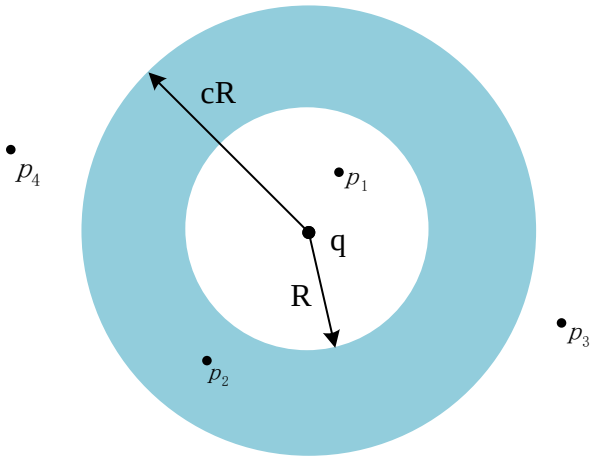

图 3.6　离散空间上的ANN

我们需要让在相近的 R 近邻（图 3.6 中白色部分）进行 hash 冲突的概率更大，而 cR 近邻（图 3.6 中蓝色部分）则越小。但是原始的 LSH 则会使得 R 近邻和 cR 近邻都尽可能的大，会导致查找的结果很多时候并不能得到一个较好的回馈。

构造一个数据结构 LSH，对于任意查询$q \in \mathbb{R}^d$，如果在 P 中存在 q 的 R 近邻，能以 1-δ (δ>0) 的概率给出 P 中 q 的 cR 近邻。跟连续空间上的 LSH 相似，如果 LSH 函数的规模扩大一倍，这个概率将从 1-δ上升到 $1-\delta^2$。当我们对离散空间进行放缩，使 R=1 时，ANN 问题就变成了 c 近似最近邻问题 (c-approximate near neighbor problem, c-NN) 。

由此，我们可以正式地定义局部敏感哈希。如果对于任意两点p, q，从 H 中均匀随机选择一个函数 g，有

(a) 如果$||p-q|| \le R$，$\Pr(g(p)=g(q)) \ge P_1$；

(b) 如果$||p-q|| \ge cR$，$\Pr(g(p)=g(q)) \le P_2$；

(c) $0 < P_2 < P_1 < 1$，

则称哈希函数族 H 是局部敏感的，也称 Gapped LSH。P_1和P_2间的距离可能非常小，需要额外添加步骤放大两者的间隙。

我们选择l个函数$h_1, h_2, \dots, h_l$，其中$h_j(q) = \left(g_{j1}(q), \dots, g_{jk}(q)\right)$，而$g_{ji}$是从 H 中均匀随机选择的函数 $(1 \le j \le l,\ 1 \le i \le k)$ 。使用这些函数将 P 转换到l个散列表中，之后对于每一组询问q，查找这些散列表以提取数据点进行验证。如果$||p-q|| \le R$，$\Pr\left(h_j(p)=h_j(q)\right) \ge {P_1}^k$；如果$||p-q|| \ge cR$，$\Pr\left(h_j(p)=h_j(q)\right) \le {P_2}^k$。

对于一个 c-NN 问题，我们先找到 $L = tl$ 个点验证，然后中止。定义事件 E1 为假阳性的数量少于 $L = tl$ 个数据点（即总哈希冲突次数$< tl$），E2 为以 1-θ的概率找到一

个解决方案。我们需要根据给定的t和θ，找到最佳参数l和k。

$||p-q|| \geq cR$碰撞的概率$\Pr\left(h_j(p)=h_j(q)\right) \leq {P_2}^k = \frac{1}{n}$（n 表示数据点的数量），因此可以推导得$k = -\frac{\ln n}{\ln P_2}$。

q 在散列表 a 中发生冲突的期望 E(#collisions with q in a table) ≤ 1 ，所以l个散列表中 q 发生冲突的期望 E(total #collisions with q in l tables) ≤ l 。根据马尔可夫不等式 (Markov Inequality) 可知，Y 为仅假设在非负值上的随机变量，对于任意t ∈ $\mathbb{R}^+$，都有$\Pr(Y \geq t) \leq \frac{E(Y)}{t}$。因此，

$$\Pr(\text{total \#collisions} \geq tl) \leq \frac{l}{tl} = \frac{1}{t} \tag{3.29}$$

$$\Pr(< tl \text{ collisions}) \geq 1 - \frac{1}{t} \tag{3.30}$$

如果存在 NN（即$||p-q|| \leq R$），找到一个 ANN 的概率为

$$\begin{aligned} &\Pr\left(h_1(p)=h_1(q) \vee \ldots \vee h_l(p)=h_l(q)\right) \\ &= 1 - \Pr\left(h_1(p) \neq h_1(q) \wedge \ldots \wedge h_l(p) \neq h_l(q)\right) \\ &\geq 1 - \left(1-{P_1}^k\right)^l \approx 1 - e^{-l{P_1}^k} = 1 - \theta \end{aligned} \tag{3.31}$$

在$\theta = e^{-l{P_1}^k}$，${P_2}^k = \frac{1}{n}$的条件下，

$$l = -\ln\theta / {P_1}^k = -\ln\theta / n^{-\frac{\ln P_1}{\ln P_2}} = -\ln\theta \times n^{\frac{\ln P_2}{\ln P_1}} = O(n^\rho)(\rho = \frac{\ln P_2}{\ln P_1} < 1) \tag{3.32}$$

根据上述分析，可知

$$\Pr(E1 \cap E2) \geq 1 - (1 - \Pr(E1)) - (1 - \Pr(E2)) = 1 - (\frac{1}{t} + \theta) \tag{3.33}$$

令$\delta = (\frac{1}{t} + \theta)$，事件 E1 和 E2 同时为真的概率就是$\Pr(E1 \cap E2) \geq 1 - \delta$。$\Pr(E1 \cap E2) = 1 - \delta$当且仅当所有假阳性情况都被找到，$\Pr(E1) = 1$。算法空间复杂度$O(dn + nl) = O(dn + n^{1+\rho})$，搜索部分的时间复杂度$O(dl) = O(dn^\rho)$。

对于一个 R 近邻问题，我们在l个散列表中搜索询问串 q 的哈希值，合并所有产生哈希冲突的项，对它们进行验证。关于假阳性和敏感度部分的分析与 c-NN 相同，空间复杂度也是一样的。而在搜索部分的时间复杂度上，需要额外考虑解个数的期望，E(#false positives + #occurrences of solutions)=$O(dl) + O\left(d \times {P_1}^k \times l \times \#solutions\right)$。由于${P_1}^k \times l = n^{-\rho} \times n^\rho = 1$，所以化简得$O(dn^\rho + d \times \#solutions)$。

对于一个长度为 d 的子串，

(a) 如果$||p-q|| \leq R$，$\Pr\left(g(p)=g(q)\right) \geq \frac{d-r}{d} = P_1$；

(b) 如果$||p-q|| \geq cR$，$\Pr(g(p)=g(q)) \leq \frac{d-cr}{d} = P_2$；

所以$\rho = \frac{\ln P_2}{\ln P_1} = \frac{\ln 1-\frac{r}{d}}{\ln 1-\frac{cr}{d}} \leq \frac{1}{c}$，$l = O(n^{\rho}) = O\left(n^{\frac{1}{c}}\right)$。

3.3.2 Order Min Hash 模型

设$\mathcal{H}$是定义在集合$\mathcal{U}$（全集）上的散列函数族。当

$$s(x,y) \geq s_1 \Rightarrow Pr_{\mathfrak{h}\in\mathcal{H}}[\mathfrak{h}(x)=\mathfrak{h}(y)] \geq p_1, \tag{3.34}$$

$$s(x,y) \leq s_2 \Rightarrow Pr_{\mathfrak{h}\in\mathcal{H}}[\mathfrak{h}(x)=\mathfrak{h}(y)] \leq p_2, \tag{3.35}$$

则称集合$\mathcal{H}$上的概率分布对相似度$\mathcal{s}$关于(s_1, s_2, p_1, p_2)敏感的。其中$s_1 \geq s_2$，$p_1 \geq p_2$。如果存在一组散列函数的分布关于 (s_1, s_2, p_1, p_2) 敏感，则允许使用 Gapped LSH 对相似序列进行聚类。在上面的定义中，具体概率取决于$\mathcal{H}$中对任意 x, y∈ $\mathcal{U}$ 构造的哈希函数的选择。在 Gapped LSH 中，相似元素之间哈希冲突的概率上升 $(\geq p_1)$ ，而对于不同元素，哈希冲突的概率较小 $(\leq p_2)$ 。

编辑相似度上的 LSH 要求必须对字符串的 k-mer 内容和相对顺序都敏感，但对于 k-mers 在字符串中的绝对位置相对不敏感。这引出了下面的定义。与 minHash 类似，k-mers 是通过在 k-mers 上使用置换来随机选择的。为了保留有关的相对顺序的信息，我们并非只选取 1 个最小的哈希值，而是l个。取l个最小的 k-mers 的散列值，按照它们在初始序列中出现的顺序（而不是随机置换所定义的顺序）记录。

此外，该方法必须处理重复的 k-mers。同一 k-mer 的两个副本出现在序列中的不同位置，区分这两个副本对于 k-mer 之间的相对顺序很重要。我们通过在 k-mers 后附加“出现次数”，使其唯一。

更准确地说，对于长度$|S| = n$的字符串 S，考虑 k-mers 对及其出现次数的集合$\mathcal{M}_k^w(S)$。如果序列 S 中有 x 个 m 的副本，那么集合$\mathcal{M}_k^w(S)$中有 x 对元素，形如$(m,0), \ldots, (m, x-1)$。副本的出现次数表示在序列 S 中，该特定副本的左侧序列中存在的其它副本 m 的数量。也就是说，如果 m 是 S 中位置在 i 处的 k-mer（即$m = S[i:k]$），它的出现次数是$|\{j \in [i] | S[j:k] = m\}|$。该集合是关于字符串 S 的 k-mer 的“多重集合”，或者叫 k-mers 的“加权集合”，其中出现次数就是 k-mer 的权重（因此有上标 w）。我们把 k-mer 的 (m, i) 对和出现数称为“对应”的 k-mer。

$\sum^k \times [n]$上的一个置换π定义了两个函数$\mathfrak{h}_{l,\pi}^w$和$\mathfrak{h}_{l,\pi}$。$\mathfrak{h}_{l,\pi}^w(S) = ((m_1, o_1), \ldots, (m_l, o_l))$是一个长度为 l，由项 $\mathcal{M}_k^w(S)$ 组成的向量。具体要求如下：

- 根据置换π，对(m_i, o_i)是最小的 l 个 $\mathcal{M}_k^w(S)$ 中的项；
- 按 k-mer 在序列 S 中出现的相对顺序，在向量中列出这些对。也就是说，如果 i < j， $m_i = S[x{:}\,k]$ 且 $m_j = S[y{:}\,k]$，那么就有x < y。

向量 $\mathfrak{h}_{l,\pi}(S) = (m_1, \dots, m_l)$ 只包含来自 $\mathfrak{h}_{l,\pi}^w(S)$ 的 k-mers，并以如上顺序排列。OMH 方法定义为哈希函数集 $\mathcal{H}_{k,l} = \{\mathfrak{h}_{l,\pi} | \sum^k \times [n] \textit{上的置换} \, \pi\}$ 上的均匀分布[23]。

对于极端情况，当$l = \mathrm{n} - \mathrm{k} + 1$时，向量包含覆盖整个序列 S 的所有加权 k-mers。在这种情况下，OMH 值的相等意味着序列的完全一致。

在另一种极端情况下，当$l = 1$时，向量仅包含一个 k-mer，因此不存在相对顺序信息。在这种情况下，只会考虑S_1和S_2之间的 k-mer 内容相似性，即 MinHash。

两个序列的加权 Jaccard 相似度 $J^w(S_1, S_2)$ 是其 k-mer 的加权 Jaccard。由于 k-mers 在 $\mathcal{M}_k^w(S)$ 中的出现次数是唯一的，因此加权 Jaccard 相似度等效定义为$J^w(S_1, S_2) = J(\mathcal{M}_k^w(S_1), \mathcal{M}_k^w(S_2))$。

因为$\ell = 1$时，OMH 等价于 MinHash；$l = \mathrm{n} - \mathrm{k} + 1$时，哈希又失去意义。因此在下文的讨论中，我们规定$l \in [2, \mathrm{n} - \mathrm{k}]$。因此，我们需要证明对于任意 $\ell \in [2, \mathrm{n} - \mathrm{k}]$，$1 > \mathrm{s}_1 \geq \mathrm{s}_2 > 0$，存在函数 $\mathrm{p}_{\mathrm{n,k,l}}^1(\cdot)$ 和 $\mathrm{p}_{\mathrm{n,k,l}}^2(\cdot)$，OMH 在编辑距离上对 $\left(\mathrm{s}_1, \mathrm{s}_2, \mathrm{p}_{\mathrm{n,k,l}}^1(\cdot), \mathrm{p}_{\mathrm{n,k,l}}^2(\cdot)\right)$ 敏感。

假设S_1和S_2是两个长度为 n 的序列，每个序列中 k-mer 的数量n_k=n-k+1。当 n 非正(n≤0) 时，二次项系数$\binom{\mathrm{n}}{\mathrm{k}}$=0，表示从空集中选择元素的方案数为 0。

令$E_s(\mathrm{S}_1, \mathrm{S}_2) \geq \mathrm{s}_1$，则有$E_d(\mathrm{S}_1, \mathrm{S}_2) = 1 - E_s(\mathrm{S}_1, \mathrm{S}_2) \leq 1 - \mathrm{s}_1$。因此编辑距离$\leq n(1 - \mathrm{s}_1)$。因为一位不匹配或是编辑操作最多影响 k 个 k-mers，所以可以推出能匹配上的的 k-mer 的数量最少为$n_k - kn(1 - \mathrm{s}_1)$。

当两个序列中所有没匹配上的 k-mers 都互不相同时，并集$\mathcal{M}_k^w(S_1) \cup \mathcal{M}_k^w(S_2)$的大小达到最大。所以有$|\mathcal{M}_k^w(S_1) \cup \mathcal{M}_k^w(S_2)| \leq n_k + kn(1 - \mathrm{s}_1)$。

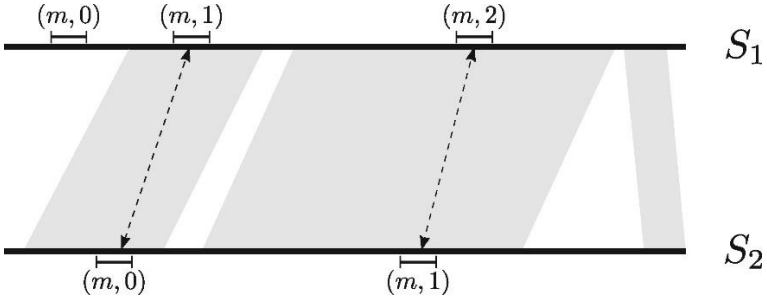

图 3.7 S_1和S_2匹配示意图[23]

我们可以根据匹配中“对应的”的 k-mer 的数量来估计哈希冲突的概率。一个“对应的”的 k-mer 由 m 和它的出现次数组成。匹配成功的 k-mer 对的出现次数可能不一致（如图 3.7 所示）。对于每一个 m 存在这种出现次数不一致的匹配，意味着匹配不成功的 k-mer 中存在包含 m 的对（如图 3.7 中S_1中的(m,0)）。因此，任意置换只要满足(m,0)在最小的l个“对应的”的 k-mer 中，就不会导致哈希冲突。

设S_1和S_2中出现次数不同的k-mers的数量为x，因此最少可以产生x+k-1位不匹配。而因为$E_s(\mathrm{S}_1,\mathrm{S}_2) \geq \mathrm{s}_1$，不匹配的位数最多为$2n(1-\mathrm{s}_1)$（只采取增删操作）。所以，$\mathrm{x}+\mathrm{k}-1 \leq 2n(1-\mathrm{s}_1)$，可得$\mathrm{x} \leq 2n(1-\mathrm{s}_1)-k+1$。而序列加权 k-mer 交集大小可以视作 m 相同的 k-mer 数量减去 m 相同 o 不同的 k-mer 数量。

$$\begin{aligned}|\mathcal{M}_k^w(S_1) \cap \mathcal{M}_k^w(S_2)| &\geq n_k - kn(1-\mathrm{s}_1) - x \\ &\geq \mathrm{n}-\mathrm{k}+1-kn(1-\mathrm{s}_1)-2n(1-\mathrm{s}_1)+k-1 \\ &= n-n(k+2)(1-\mathrm{s}_1)\end{aligned} \tag{3.36}$$

通过置换π，$\mathcal{M}_k^w(S_1) \cup \mathcal{M}_k^w(S_2)$中的每一项都有相同的概率被选入最小的$l$项中。因此，当$E_s(\mathrm{S}_1,\mathrm{S}_2) \geq \mathrm{s}_1$时，产生一次哈希冲突的概率

$$\begin{aligned}Pr\left[\mathfrak{h}_{l,\pi}(S_1) = \mathfrak{h}_{l,\pi}(S_2)\right] &\geq Pr\left[\mathfrak{h}_{l,\pi}^w(S_1) = \mathfrak{h}_{l,\pi}^w(S_2)\right] \\ &\geq \frac{\binom{n-n(k+2)(1-\mathrm{s}_1)}{l}}{\binom{n_k+kn(1-\mathrm{s}_1)}{l}}\end{aligned} \tag{3.37}$$

这里找到了式 3.34 中s_1和p_1的关系，因此 OMH 是满足式 3.34 的。

因为$\mathfrak{h}_{l,\pi}^w(S)$中所有元素都是唯一的，所以我们可以使用集合$\{\mathfrak{h}_{l,\pi}^w(S)\}$表示向量$\mathfrak{h}_{l,\pi}^w(S)$中的所有元素。令事件 C 表示$\{\mathfrak{h}_{l,\pi}^w(S_1)\}$=$\{\mathfrak{h}_{l,\pi}^w(S_2)\}$。在事件 C 发生的条件下，可以得到$\mathfrak{h}_{l,\pi}(S_1) = \mathfrak{h}_{l,\pi}(S_2) \Leftrightarrow \mathfrak{h}_{l,\pi}^w(S_1) = \mathfrak{h}_{l,\pi}^w(S_2)$。因为 $\mathfrak{h}$ 是 $\mathfrak{h}^w$ 只使用 k-mer 的特殊情况，所以⇐永远成立。而由于$\{\mathfrak{h}_{l,\pi}^w(S_1)\}$=$\{\mathfrak{h}_{l,\pi}^w(S_2)\}$，加权向量 $\mathfrak{h}^w$ 中的内容是相同的。k-mers 是按照它们在各自序列中出现的顺序列出，同时对于相同的 k-mer，也可以视作按照它们的出现次数列出，所以未加权向量 $\mathfrak{h}$ 的相等意味着加权向量的相等，⇒也是一定成立的。

令 m=$|\mathcal{M}_k^w(S_1) \cap \mathcal{M}_k^w(S_2)|$，相当于两个序列加权 k-mer 集交集的大小。当且仅当通过置换π后，l个最小“对应的”k-mers 全部来自$\mathcal{M}_k^w(S_1) \cap \mathcal{M}_k^w(S_2)$，事件 C 成立。因此，

$$Pr\left[\mathfrak{h}_{l,\pi}^{w}(S_1)=\mathfrak{h}_{l,\pi}^{w}(S_2)\right]=\frac{\binom{\left|\mathcal{M}_k^w(S_1)\cap\mathcal{M}_k^w(S_2)\right|}{l}}{\binom{\left|\mathcal{M}_k^w(S_1)\cup\mathcal{M}_k^w(S_2)\right|}{l}}\leq\frac{\binom{m}{l}}{\binom{n_k}{l}} \tag{3.38}$$

现在我们考虑在$\mathcal{M}_k^w(S_1)\cap\mathcal{M}_k^w(S_2)$中的$\mathcal{M}_k^w(S_1)$和$\mathcal{M}_k^w(S_2)$分别按照它们在$S_1$和$S_2$中的出现顺序列出，长度都是 m。在条件 C 下，事件$\mathfrak{h}_{l,\pi}(S_1)=\mathfrak{h}_{l,\pi}(S_2)$相当于通过哈希函数$\mathfrak{h}_{l,\pi}^{w}$在$\mathcal{M}_k^w(S_1)$和$\mathcal{M}_k^w(S_2)$中选择了长度为$l$的公共子序列。由于集合中的元素没有重复，因此$\mathcal{M}_k^w(S_1)$和$\mathcal{M}_k^w(S_2)$中的公共子序列 (Common Subsequence , CS) 问题等价于在长度为 m 的整数序列中寻找递增子序列 (Increasing Subsequence, IS) 的问题。

$$Pr\left[\mathfrak{h}_{l,\pi}(S_1)=\mathfrak{h}_{l,\pi}(S_2)|(C)\right]\leq\max_{\pi\in[m]!}Pr[pick\ \ IS\ \ of\ \ length\ \ l\ \ in\ \ \pi([m])] \tag{3.39}$$

其中 $[m]!$ 代表 $[m]$ 的全排列。

当$i,n,l\in\mathbb{N}$，$n\geq i\geq l$，对于任意序列长度为 n 且最长上升子序列 (Longest Increasing Subsequence, LIS) 长度为 i 的序列来说，长度为 l 的递增子序列的最大数量为

$$\left(\frac{n}{i}\right)^{l}\binom{i}{l} \tag{3.40}$$

并且这是个紧密边界 (tight bound) 。证明如下。

首先需要引入一种排序算法的概念，称为耐心排序 (patience sorting) 。给出一叠上标数字的纸牌，打乱顺序后一张一张放入堆栈中。只有当现在放入的纸牌上的数字小于某一堆栈栈顶数字时，它才能被放入当前栈顶；如果不存在这样的堆栈，就在右侧新建一个堆栈。要求在所有纸牌放置完成后，最小化桌面上的堆栈的数量。例如，有 5 张纸牌，牌上的数字打乱后分别为 7，2，8，1 和 3。

图 3.8　耐心排序示意图

贪心的做法被证明是可行的。即尽可能把牌往靠左的堆栈上放，这样得到的所有堆栈的栈顶组成的序列就是其中一个最长上升子序列。牌堆具有如下两个属性：

(a) 每个堆栈的栈顶组成的序列是递增的。假设按照贪心思想放置牌，会出现栈顶序列不递增的情况，且第一次出现在放置牌 X 时。那么牌 X 左侧的栈顶元素 Y＞X，按

照贪心思想，X 至少可以被放置在 Y 所在的栈顶，与假设的前提相悖，所以假设不成立。

(b) 堆栈的数量等于 LIS 的长度。原始牌组的任意上升子序列中，不可能存在两张牌在同一个堆栈。假设存在上升子序列 a，其中两个元素被放置在同一个堆栈中。因为堆栈中元素从栈底到栈顶是单调递减的，所以两个元素中较小的元素一定比较大元素后被放入堆栈中，a 不可能是上升子序列。因此，任意上升子序列最多从每堆中选取一个元素。

当从每个堆栈中取一个元素，且组成上升序列时，这个序列必然是最长上升子序列。因为原序列所有元素都在堆栈中。假设有比这个更长的最长上升子序列 b，说明存在两个或以上元素来自同一堆栈。而由上述分析，b 就不可能是上升序列。

设$i' \in [l, i]$，有一个长度为 n 的序列 S，其中 LIS 长度为i'。经过耐心排序后，每个堆栈的高度组成向量$\mathrm{s} = (s_0, \dots, s_{i'-1})$。S 中长度为$l$的上升子序列数量的上界

$$\mathrm{g(s)} = \sum_{\substack{A \subset [i'] \\ |A| = l}} \prod_{j \in A} s_j \tag{3.41}$$

因为从每个堆栈中取一个元素不一定就能组成上升序列，所以g(s)是一个比较宽松的上界。

当$s_0 = \cdots = s_{i'-1} = n/i'$时，g(s)取得最大值。集合$\mathrm{C} = \left\{\mathrm{s} = (s_0, \dots, s_{i'-1}) | \sum_j s_j = n\right\}$是一个紧集 (compact set) 。有界定理表明连续函数在紧集上是有界的，并且在集合上的某些点取得最大值与最小值。因此g(s)在 C 上存在最大值。假设在 s 中，存在不是所有s_j都相等。那么我们不失一般性地设$s_{i'-1}$和$s_{i'-2}$不相等。令$\alpha = (s_{i'-1} + s_{i'-2})/2$，向量$\mathrm{s}' = (s_0, \dots, s_{i'-3}, \alpha, \alpha)$。

$$\rho(\mathrm{x}) = \sum_{\substack{A \subset [i'-2] \\ |A| = x}} \prod_{j \in A} s_j \tag{3.42}$$

然后我们从$\mathrm{g}'(\mathrm{s})$分解出既不包含$s_{i'-1}$，又不包含$s_{i'-2}$的项（=$\rho(l)$），包含$s_{i'-1}$或$s_{i'-2}$其中一个的项（=$2\alpha\rho(l-1)$），包含$s_{i'-1}$和$s_{i'-2}$的项（=$\alpha^2\rho(l-1)$）。因为 n 个正数的算术平均数不小于它们的几何平均数。当且仅当这 n 个正数都相等时，它们的算术平均数和几何平均数的值相等。所以有$\alpha^2 > s_{i'-1}s_{i'-2}$。

$$\begin{aligned} \mathrm{g}'(\mathrm{s}) &= \rho(l) + 2\alpha\rho(l-1) + \alpha^2\rho(l-1) \\ &> \rho(l) + (s_{i'-1} + s_{i'-2})\rho(l-1) + s_{i'-1}s_{i'-2}\rho(l-1) \\ &= \mathrm{g(s)} \end{aligned} \tag{3.43}$$

所以当s中包含两个值不同的元素时，g(s)就无法达到最大值。只有当$s_0 = \cdots =$

$s_{i'-1}=n/i'$时，g(s)取得最大值。

$$\max_{s\in C}\mathrm{g(s)}=\sum_{A\subset[i],|A|=l}(\frac{n}{i'})^l=(\frac{n}{i'})^l\binom{i'}{l}\triangleq m_{n,l}(i') \tag{3.44}$$

函数$m_{n,l}(\cdot)$是一个增函数。当$i'=i$的时候，函数取得最大值。

我们考虑这样一个序列$\mathrm{S}(i,n)$，其中 n 能被 i 整除。序列具体形式如下。

$$\underbrace{\left(\frac{n}{i}-1\right)\dots 0}_{block\ 1}\underbrace{\left(\frac{2n}{i}-1\right)\dots(\frac{n}{i})}_{block\ 2}\dots\underbrace{(n-1)\dots(n-\frac{n}{i})}_{block\ i} \tag{3.45}$$

每个块的长度为 n/i ，每个块中的数字按降序排列，但块的开头按升序排列。块 j 由一个上升序列 $(jn/i-1)\dots((j-1)n/i)$ 组成。

当耐心排序算法应用于序列 $\mathrm{S}(i,n)$ 时，堆栈从下到上、从左到右逐个填充，并且具有相同的 n/i 的高度。因此，从每个堆栈中任意选择一个元素都是 $\mathrm{S}(i,n)$ 的有效上升子序列，并获得等式 3.44 中的界。所以式 3.40 得以证明。

我们最终需要证明式 3.35。即找到一个p_2，使得$E_s(\mathrm{S}_1,\mathrm{S}_2)\le \mathrm{s}_2\Rightarrow Pr\left[\mathfrak{h}_{l,\pi}(S_1)=\mathfrak{h}_{l,\pi}(S_2)\right]\le p_2$。依照上文的分析，

$$\begin{aligned}&Pr\left[\mathfrak{h}_{l,\pi}(S_1)=\mathfrak{h}_{l,\pi}(S_2)\right]\\&=Pr\left[\mathfrak{h}_{l,\pi}(S_1)=\mathfrak{h}_{l,\pi}(S_2)|\left\{\mathfrak{h}^w_{l,\pi}(S_1)\right\}=\left\{\mathfrak{h}^w_{l,\pi}(S_2)\right\}\right]\\&\quad\cdot Pr\left[\left\{\mathfrak{h}^w_{l,\pi}(S_1)\right\}=\left\{\mathfrak{h}^w_{l,\pi}(S_2)\right\}\right]\end{aligned} \tag{3.46}$$

根据式 3.38 和式 3.39，以及式 3.40，最后可以推出

$$Pr\left[\mathfrak{h}_{l,\pi}(S_1)=\mathfrak{h}_{l,\pi}(S_2)\right]\le\max_{\substack{l\le m\\ m\le n_k}}\frac{(\frac{m}{L})^l(\frac{L}{l})}{\binom{m}{l}}\frac{\binom{m}{l}}{\binom{n_k}{l}}\le\frac{(\frac{n_k}{L})^l(\frac{L}{l})}{\binom{n_k}{l}} \tag{3.47}$$

其中 L 表示$\mathcal{M}^w_k(S_1)$和$\mathcal{M}^w_k(S_2)$的最长公共子序列的长度。式 3.48 右侧的式子是关于 L 的增函数。当 $\mathrm{L}<l$ 时，式子等价于 0；当 $\mathrm{L}=\mathrm{n}-\mathrm{k}+1$ 时，式子等价于 1。

给定$E_s(\mathrm{S}_1,\mathrm{S}_2)\le \mathrm{s}_2$的条件，有$\mathrm{s}_2\ge E_s(\mathrm{S}_1,\mathrm{S}_2)\ge(L+k-1)/n$。$\mathrm{L}\le \mathrm{ns}_2-k+1$，将式 3.48 中的 L 用$\mathrm{ns}_2-k+1$替换，得

$$Pr\left[\mathfrak{h}_{l,\pi}(S_1)=\mathfrak{h}_{l,\pi}(S_2)\right]\le\frac{(\frac{n_k}{\mathrm{ns}_2-k+1})^l\binom{\mathrm{ns}_2-k+1}{l}}{\binom{n_k}{l}} \tag{3.48}$$

综上，当$p_2=\frac{(\frac{n_k}{\mathrm{ns}_2-k+1})^l\binom{\mathrm{ns}_2-k+1}{l}}{\binom{n_k}{l}}$时，$E_s(\mathrm{S}_1,\mathrm{S}_2)\le \mathrm{s}_2\Rightarrow Pr\left[\mathfrak{h}_{l,\pi}(S_1)=\mathfrak{h}_{l,\pi}(S_2)\right]\le p_2$。找到了式 3.35 中$\mathrm{s}_2$和$p_2$的关系，因此 OMH 是满足式 3.35 的。所以，OMH 是关于$\left(\mathrm{s}_1,\mathrm{s}_2,\mathrm{p}^1_{\mathrm{n,k,l}}(\mathrm{s}_1),\mathrm{p}^2_{\mathrm{n,k,l}}(\mathrm{s}_2)\right)$敏感的，可以被用于相似序列的聚类。

3.3.3 模型复现

作为预处理步骤，需要把单个蛋白质序列划分为 k-mer。并根据 OMH 算法的特点，最终“对应的”k-mer 包括 k-mer 和它当前在序列中的出现次数。

算法 3.1　k-mer 分割算法

输入：长度为 L 的蛋白质序列 S，k-mer 的长度 k
输出：输入序列 S 的全部 k-mer 加权集合 V

```
V ← [ ]                                  //初始化集合
for i = 1 to L-k+1                       //每条序列拥有的 k-mer 数量为 L-k+1
    s ← S[i..i+k-1]
    V ← V ∪ [s, H[s]]
    H[s] = H[s]+1
end for
return V
```

算法 3.1 中，重点在于对 k-mer 出现次数的统计，可以用 c++中内置的 unordered_map 关键词实现。序列 S 中 k-mer 的截取，也可以采用字符串 string 的内置函数 substr，基于滑动窗口算法思想进行一遍线性扫描。加权集合 V 的元素类型可以设置为 pair<string, int>，方便对元素进行打包。

接下来，就是 OMH 算法的具体实现部分。

算法 3.2　Order Min Hash 算法

输入：n 组蛋白质序列 S[]，k-mer 的长度 k，散列函数模数 p，全域散列模数 pp
输出：聚类后存在同源性可能的序列对集合 V

```
V ← [ ]
let A[1..L] be a new array               //A 存放散列函数的基数
for i=1 to L
    A[i] = RANDOM(1, p-1)                //生成 L 组散列函数
end for
for ii = 1 to L
    for i = 1 to n
        m ← |k_mer[i]|                   //m 是第 i 个序列 k-mer 的数量
        for j = 1 to m
            x ← k_mer[i][j]
            val = 1                      //val 用来计算 k-mer 的散列值
            for jj = 1 to k
```

```
13:            val = ( val * A[ii] + x[jj] ) % p
14:          end for
15:          hash[j] = val % pp
16:       end for
17:       select hash[1..m] corresponding to the smallest l values
          //将最小的 l 个值放置在 hash 序列最左边，l个值内部保持原有序列排序
18:       for j = 1 to l
19:          v ← v ∪ hash[j]
20:       end for
21:       set[v] ← set[v] ∪ i      //对包含l个值和相对位置信息的列表做一个散列
22:    end for
23:    for i = 1 to |set[ ]|
24:       for j = 1 to |set[i]|
25:          for jj = j + 1 to |set[i]|
26:             V ← V ∪ ( set[i][j] , set[i][jj] )      //散列相同的两个序列可能同源
27:          end for
28:       end for
29:    end for
30: end for
31: return V
```

简化 OMH 算法的实现，需要对 c++的标准模板库 (Standard Template Library, STL) 具有一定掌握。尤其是将其中 l 个 k-mer 组的序列表映射到一个集合上，基础数据结构较难实现。以下给出我的 OMH 函数具体实现过程。

代码清单 3.1　OMH 函数

```
vector< pair<string, int> > k_mer[MAXN];
unordered_map<string, int> mp, mpp;
map< vector<string>, int > hash_table, emptyh;
vector<int> hashset[MAXN], emptyhs;
set< pair<int, int> > preset;
void omh() {
    for(int ii = 0; ii < L; ii ++) {                    //L组hash函数
        hash_table = emptyh;    num = 1;
        for(int i = 0; i <= num; i ++)    hashset[i] = emptyhs;
        for(int i = 0; i < n; i ++) {
            int m = k_mer[i].size();
```

```
                for(int j = 0; j < m; j ++) {
                    string x = k_mer[i][j].first;    long long h = 1;
                    for(int jj = 0; jj < k; jj ++)    h = (h * hasha[ii] + x[jj]) % p;
                    hashval[j].val = h % pp;    hashval[j].no = j;
                }
                //找出最小l个hash值对应的k-mer
                sort(hashval, hashval + m, cmp);    sort(hashval, hashval + l, cmp2);
                vector<string> v;
                for(int j = 0; j < l; j ++)    v.push_back(k_mer[i][hashval[j].no].first);
                if(hash_table[v] == 0)    hash_table[v] = num ++;
                hashset[hash_table[v]].push_back(i);
            }
            for(int j = 1; j < num; j ++) {
                int ans = hashset[j].size();
                for(int jj = 0; jj < ans; jj ++)
                    for(int jjj = jj + 1; jjj < ans; jjj ++)
                        preset.insert ( make_pair(hashset[j][jj], hashset[j][jjj] ) ) ;
            }
        }
    }
```

在代码清单 3.1 中，将 L 个存储 l 个 k-mer 序列的散列表巧妙地转换成了，先将 l 个 k-mer 序列映射成某个 hash 值，再将 l 个 k-mer 序列对应的原始序列放入 hash 值对应的集合中。最后将这些集合里的所有序列两两配对，存入目标集合中。

而因为既要选出 l 个最小哈希值，又要保持选出的 k-mer 值相对顺序不改变，我们采用两遍排序，但两组排序的数据范围并不相同。第一遍按照 hash 值对结构体 hashval[0..m-1]进行排序，第二遍按照记录的原始位置对结构体 hashval[0..l-1]进行排序，以符合算法要求。

最后，只需要将待检验集合中的序列对两两进行相似度计算，就能得到最后的结果。相似度计算采用动态规划的思想求解编辑距离，如算法 3.3 所示。

算法 3.3　动态规划求解编辑相似度

输入：长度分别为 l_1, l_2 的蛋白质序列 S_1 和 S_2

输出：输入序列 S_1 和 S_2 的编辑相似度 s

1: let dp[0..l_1][0..l_2] be a new array

```
2:  for i = 0 to  l1-1
3:      dp[i][0] = i
4:  end for
5:  for i = 0 to  l2-1
6:      dp[0][i] = i
7:  end for
8:  for i = 1 to  l1
9:      for j = 1 to  l2
10:         if  S1[i] =  S2[j]
11:             dp[i][j] = dp[i-1][j-1]
12:         else dp[i][j] = 1 + min{ dp[i-1][j], dp[i][j-1], dp[i-1][j-1] }
13:     end for
14: end for
15: s = 1 – dp[l1][l2] / max(l1, l2)
15: return s
```

3.4 基于 MinHash 算法和 LSH 算法的高效近似 Jaccard 相似度估计算法

对 OMH 算法进行分析，我们不难发现，因为 OMH 的提出是用来对碱基序列的相似度进行聚类的，所以它的一些实现在蛋白质序列上并不实用。具体来说，因为碱基序列只包含 ACGT 四种字符，假设 k-mer 中的 k 为 3，也就只能产生 $4^3 = 64$ 种 k-mer。如果序列长度比较长，一个序列就非常容易产生多个相同的 k-mer。所以 OMH 中提出的包含出现相对位置的“对应的”k-mer 就很巧妙地解决了加权集合的问题。

但是蛋白质序列中含有 26 个字符，假设 k-mer 中的 k 为 3，就能产生 $26^3 = 17576$ 种 k-mer。这个数量级是非常恐怖的，意味着一个长度为 L 的序列中出现多个相同的 k-mer 的概率会比碱基序列低很多。

因此 OMH 关于相对位置的考虑就显得不太必要。将 OMH 算法进行改造，使得哈希冲突率在可接受范围内上升，期望能找到效率和准确性的更优平衡点。

本文模型吸收 OMH 模型关于 l 个最小哈希值的思想，一定程度上会同时降低 TP 率和 FP 率，但是能很大程度优化时间复杂度。最后回归 LSH 算法，仅按照l 个最小哈希值进行聚类，改进了 OMH 模型哈希冲突率较低的特点。

3.5 本章小结

本章旨在详述本文提出的蛋白质同源性搜索的高效算法。首先从最长公共子序列

LCS 问题入手，引出 Jaccard 相似度，证明 Jaccard 相似度是编辑相似度的上界，并利用该上界实现低同源性序列的快速过滤。接着从算法时间复杂度角度进行论证，寻找问题规模降维的可行性。之后，引入 Order Min Hash 模型，先对文本相似度算法和散列函数交叉领域的术语做了详细介绍，分析 MinHash 算法和 Locality-sensitive hashing 算法的原理，并具体计算了概率和期望。对于 Order Min Hash 模型的理念，给出了基于 C++的实现方法。最后，基于 MinHash 算法和 Locality-sensitive hashing 算法，对 Order Min Hash 模型进行修改，给出 Jaccard 相似度的高效近似估计算法。

4 算法效率与准确性在大数据集上的评估

4.1 数据集与任务介绍

FASTQ 和 FASTA 是存储生物序列的两种常用数据格式[24]。本文所研究的蛋白质同源性搜索问题中所处理的就是其中 FASTA 的数据。

FASTA 是一种在生物信息学领域广泛应用的、用单个大写英文字母来表示核苷酸序列或氨基酸（蛋白质）序列的文本文件格式。FASTA 格式主要是由 William R. Pearson 和 David J. Lipman 等人共同发明的[25]。所以其也可以被称之为 Pearson 格式。在 FASTA 中，一条序列由一行序列标记和后续的一行或多行序列信息组成。其中序列标记使用‘>’作为起始符号，接着是序列的名称，然后是序列的注释信息，换行之后是序列信息。为了保证 FASTA 文件的可读性，序列每行不超过 120 个字符，通常不超过 80 个字符。

图 4.1 为一个简单的 FASTA 文件示例。

```
>43FE01DC-D266-11EB-AD85-FBA22131C556
DGPSGIIWPOUXKXXBKWOGPMGTOPKLWZGKWPHMETKUKWHLWWJPUBATOWPBMKTTJKSTOHWWSIULPXHGGLOLSIGB
>43FE1B54-D266-11EB-ADF9-030D101C36E9
FBBHTFJSVMHJJJSTWTVXHUCMJUTTGHXWUBKXAJNJTBXJGUJTGAJTWXTKXXWSBHLSNAS
>43FE333C-D266-11EB-AE6C-7B98043C15DE
GPVFJCTGKJKWWTHGAMBXIIGTVVPGIJLSWMTZGMOJOHJGJGGTGBXISWKBHWTWWTGTHKZICTSIXMJBI
```

图 4.1　FASTA文件示例

需要注意的是，在 FASTA 格式中，没有对单条序列的长度区间进行限制。但与核苷酸序列碱基数跨度巨大不同，蛋白质序列最长仅含几千个氨基酸，因此在读取时还是较为方便的。一般来说，氨基酸序列包含了 23 个字母和 3 个特殊字符，见表 4.1。为了方便进行数据处理，我们只将其看作 26 个字母的排列组合。实际在具体应用中，需要考虑 B、J、X 和 Z 四个字母的特殊性。

表 4.1　FASTA支持的氨基酸编码

编码	含义	编码	含义	编码	含义	编码	含义
A	丙氨酸	H	组氨酸	O	吡咯赖氨酸	V	缬氨酸
B	D或N	I	异亮氨酸	P	脯氨酸	W	色氨酸
C	半胱氨酸	J	L或I	Q	谷氨酰胺	X	任何氨基酸
D	天冬氨酸	K	赖氨酸	R	精氨酸	Y	酪氨酸
E	谷氨酸	L	亮氨酸	S	丝氨酸	Z	E或Q
F	苯丙氨酸	M	蛋氨酸	T	苏氨酸	*	翻译停止
G	甘氨酸	N	天冬酰胺	U	硒代半胱氨酸	-	不定长间隙

本文测试数据集来源于。。。包含 90 个大小为 1.33GB 的 FASTA 文件，每个文件包含约为 1E7 条蛋白质序列，每条蛋白质序列平均约含 200 个字符。而由于同源蛋白质的稀缺性，最终找到的序列对个数应该极少。因此，在 I/O 优化的时候，重点考虑读取速度的优化。表 4.2 和表 4.3 展示了一些常见的读取方法和函数在各大平台的测试时间。

表 4.2　读取文件方案在各大平台运行时间

方法/平台/时间(s)	Linux gcc	Windows mingw	Windows VC2008
scanf	2.010	3.704	3.425
cin	6.380	64.003	19.208
cin取消流同步	2.050	6.004	19.616
fread	0.290	0.241	0.304
read	0.290	0.398	不支持
mmap	0.250	不支持	不支持
Pascal read	2.160	4.668	

表 4.3　读取字符串函数的运行效率

读取函数	所用时间
gets	72ms
fgets	76ms
scanf	960ms
getline	2189ms
cin	2275ms

我们需要对一组给定规模的蛋白质序列输入，输出其中具有同源性概率的序列对。在实验中，本文使用了两种传统的指标，效率和准确性来对用于测试这些数据集的不同方法进行评估。效率采用程序从开始到结束的运行时间来评估，运行时间越短，算法的效率就越好。而准确性的评估则是与朴素算法的运行结果进行比较。朴素算法可以保证结果的 100%准确率，从而可以找到比较算法的 FP 率和 FN 率。

4.2 实验设计

本文实验包含两部分，第一部分是在小数据集下，将本文模型同朴素算法与 OMH 模型的搜索结果进行对比；第二部分是对本文模型涉及的参数进行调整，并对细节进行优化，在大数据集下测试模型的可靠性。

在第一部分中，由于朴素算法的时间复杂度过高，所以我们将数据范围缩减至 1E4，以期在可接受的运行时间内，对三种模型的效率与准确性进行对比。为了使结果更具有

说服力，我们将 OMH 模型和本文模型的参数设置相同，并选择两组比较合理的参数分别比对，具体见表 4.4。测试数据由真实数据随机产生。考虑到 1E4 的数据范围较小，可能无法找到编辑相似度较大的序列对，为了方便结果比对，我们往里加入了 5 条在真实序列基础上进行编辑操作的序列，并把同源性的阈值设置为 0.5。选用的平台是 Windows mingw，读入均采用 getline 函数，不进行 I/O 优化。

表 4.4 实验参数设置

参数	意义	值1	值2
k	k-mer的k大小	3	4
l	选取的最小hash个数	3	2
L	hash函数个数	300	500
p	hash函数模数	19260817	19260817
pp	全域散列的最后模数	100	300

在第二部分中，为了方便比较，我们仍采用第一部分的测试数据集，在 Windows mingw 平台完成对照实验。最后将调整好参数的模型部署在 Linux 中，应用于真实的 90 组大数据集中，得到结果。

4.3 实验结果与分析

4.3.1 三种模型的效率与准确性的对比

因为朴素模型无需对参数进行调整，因此我们先对其进行测试。在 1E4 的数据集上耗时 10588.1 秒，得到了 19 组编辑相似度大于 0.5 的蛋白质序列对。

接下来我们用 10 组独立实验，分别对两组参数下的 OMH 模型和本文模型进行测试。结果如表 4.5 所示。测试结果中第一列代表找到的解的组数，第二列代表程序耗时。

表 4.5 1E4数据集下的测试结果

实验编号	OMH+ 参数值1		改进模型+ 参数值1		OMH+ 参数值2		改进模型+ 参数值2	
1	11	89.220	10	56.524	16	138.463	17	94.194
2	12	86.554	9	56.283	16	139.230	16	93.654
3	14	88.449	8	57.003	17	138.129	17	99.766
4	13	89.723	10	56.847	17	138.895	17	94.846
5	16	86.876	10	56.415	17	138.444	17	94.249
6	12	88.171	10	56.151	17	138.847	17	94.599
7	14	89.991	13	56.569	17	138.145	17	93.514
8	9	88.776	11	56.167	17	138.650	17	93.517
9	11	88.987	10	56.500	17	139.750	17	93.848
10	14	86.255	11	56.190	17	139.350	17	93.668

在第一组参数值下，OMH 模型平均能找到 12.6 个解，占解集的 66.32%，总体方差 3.64，平均用时 88.30 秒；改进模型平均能找到 10.2 个解，占解集的 53.68%，总体方差 1.56，平均用时 56.46 秒。也就是说，OMH 模型在平均准确性上优于改进模型，但存在求解不稳定的情况，但改进模型相较于 OMH 模型提速约 36.06%。

第二组参数值中，OMH 模型平均能找到 16.8 个解，占解集的 88.42%，总体方差 0.16，平均用时 138.79 秒；改进模型平均能找到 16.9 个解，占解集的 88.95%，总体方差 0.09，平均用时 94.59 秒。改进模型相较于 OMH 模型提速约 31.85%，并且两个模型都达到了很好的求解效果，不仅准确性较高，而且算法稳定性也比较好。

从算法效率来说，OMH 模型和改进模型相较于朴素算法，提升都是非常显著的。改进模型在 OMH 模型的基础上对时间复杂度进行了进一步的优化，缩小了算法时间复杂度的常数，对于大数据的改进效果会更加明显。在一些适当参数下，两者都能达到较高的准确率，但 OMH 模型的平均准确率会更好一些。

4.3.2 参数调整对改进模型的影响

实验需要调整的参数如表 4.4 所示。我们将 k=4，l=2，L=300，p=19260817，pp=100 的一组参数设置为对照组，再针对多个具体参数分别设置不同的实验组。具体结果可见表 4.6。

表 4.6　不同参数的对照实验结果

k	l	L	p	pp	解的组数	耗时
4	2	300	19260817	100	13 15	53.258 54.330
3	2	300	19260817	100	16 17	419.586 424.830
5	2	300	19260817	100	12 12	53.036 53.369
6	2	300	19260817	100	9 8	54.727 55.631
4	1	300	19260817	100	17 /	1373.520 /
4	3	300	19260817	100	7 6	50.765 52.973
4	2	500	19260817	100	16 15	91.857 92.265
4	2	700	19260817	100	16 17	142.035 149.776

续表 4.6　不同参数的对照实验结果

k	l	L	p	pp	解的组数	耗时
4	2	300	19260817	100	13	53.258
					15	54.330
4	2	300	12289	100	14	59.917
					14	59.133
4	2	300	1610612741	100	16	59.697
					16	58.088
4	2	300	19260817	300	17	59.440
					16	60.358
4	2	300	19260817	500	17	61.777
					17	61.133
4	2	300	19260817	700	15	59.799
					17	62.099

显而易见的，这五个参数对结果的影响都是显著的。具体表现在：

k 的减小意味着初筛更加宽松，能使更多序列对进入候选区域，因此会带来效率的降低和准确度的上升，增大则刚好相反。k=4 是一个比较理想的参数值，当 k 减小为 3，运行时间就扩大为原来的 8 倍左右，时间成本太大。

l=1 时，改进模型就彻底变成 MinHash，运行时间约为对照组的 26 倍，效果不理想。而l=3 时，准确率急速下降到 50%以下，在天然数据中很难保证能搜寻出解。因此l=2 也是一个相对较好的参数。

L 表示哈希函数的组数。解的组数一定是一个关于 L 的增函数，并且在 L 达到一个值后，解的组数近似于一条平滑直线。具体数值的选取应该在运行时间允许的情况下，尽量选择大一些的值。

p 的选择对结果是有一定影响的。但由于 p 的不连续性，只能说选取一个靠近2^{31}的大素数效果都还是不错的，并且带来的效率上的影响也比较小。

pp 在一定范围内的增加可以提升准确率，但超过后又会因为哈希冲突的减少而导致准确率的下降。pp=500 是一个比较理想的参数值。

4.4 本章小结

本章属于实验环节，首先对数据集与将要进行的实验任务进行介绍，接着详述了实验设计中各类参数，包括实验环境、数据集预处理过程、模型参数设置与具体的训练流程，方便其他研究者复现。再将实验结果与朴素算法的结果进行比对分析，检测算法的可信程度和实现效率。并将本篇提出的模型算法进行多次实验，其结果与该领域最新模

型结果进行对比，取得了较高的性能。最终对模型的各项参数进行优化，并进行多组对照实验，找到最优的参数。实验证明，当 k=4，l=2，L=300，p=19260817，pp=500 时，模型能取得比较理想的效果。

结　　论

针对蛋白质同源性搜索问题，即具有编辑相似性的序列对搜索问题，本文在生物大数据聚类分析的基础上，提出了基于 MinHash 算法和 LSH 算法的高效近似 Jaccard 相似度估计算法。区别于基础的 MinHash 算法，该模型在 Order Min Hash 模型的启发下，用最小l个哈希值代替最小哈希值，提高了程序的运行效率。并且，通过对 Order Min Hash 模型的改进，进一步平衡效率与准确率。同时，各参数的调整是灵活的，可以基于具体数据集的需要进行修改。

在实验结果中，我们统一以 C++语言复现了朴素算法模型和 OMH 模型，取得了与参考文献相近的结果，表明了我们实验的设计与模拟环境具有相当高的可信度。最终，以相同的数据处理，运行环境，I/O 框架和训练流程，对本文提出的蛋白质同源性搜索问题进行了多次实验。实验结果表明，我们的算法模型在 1E4 的小数据集上取得了较好的性能，效率提升迅速。

在生物数据规模快速增长的背景下，当前对蛋白质同源性搜索问题主要聚焦在聚类分析以缩小需要分析的集合大小，即主要对序列进行哈希分类。而本文提出的基于 MinHash 算法和 LSH 算法的高效近似 Jaccard 相似度估计算法在这个思路的基础上，做了进一步的分析论证，验证了这个方向的理论正确性，取得了一定程度的性能提升，或许值得更加深入地研究。