数据结构与算法分析课程设计报告

--序列对齐算法 Sequence_alignment

班级：软工 1704　　姓名：郭宇瑶　　学号：U201717031

一、问题描述

序列对齐算法

在生物信息学中，对各种生物大分子序列进行分析是一件非常基本的工作。从序列的片段测定，拼接，基因的表达分析，到 RNA 和蛋白质的结构功能预测，物种亲缘树的构建都需要进行生物分子序列相似性的比较。在遗传物质长期的演化过程中，原本相同的 DNA 序列由于其中一条序列缺失了几个片断，或增加了几个片断，或某段子序列发生了位置的变化等，从而导致他们发生了不同，这两条序列不一定能进行精确的匹配，但是他们有一定的相似度。

序列比对是一种排列 DNA，RNA 或蛋白质序列的方法，以识别可能是序列之间功能，结构或进化关系的相似区域。核苷酸或氨基酸残基的比对序列通常表示为基质内的行，在残基之间插入间隙，使得相同或相似的字符在连续的列中对齐。序列比对也用于非生物序列，例如计算自然语言或财务数据中字符串之间的编辑距离成本。

序列比对的计算方法通常分为两类：全局比对和局部比对。计算全局对齐是一种全局优化形式，它“强制”对齐以跨越所有查询序列的整个长度。相比之下，局部比对识别长序列内的相似区域，这些区域通常总体上是不同的。局部比对通常是优选的，但由于识别相似区域的额外挑战，因此可能更难以计算。已经将各种计算算法应用于序列比对问题。这些包括缓慢但正式的正确方法，如动态编程。这些还包括为大规模数据库搜索而设计的高效，启发式算法或概率方法，这些方法无法保证找到最佳匹配。

多重序列比对（Multiple sequence alignment；MSA）是对三个以上的生物学序列（biological sequence），如蛋白质序列、DNA 序列或 RNA 序列所作的序列比对。一般来说，是输入一组假定拥有演化关系的序列。从 MSA 的结果可推导出序列的同源性，而种系发生关系也可引导出这些序列共同的演化始祖。各种突变事件，例如点突变的单格变化，或是如删除突变与插入突变，可使各个序列之间产生鸿沟。MSA 常用来研究序列的保守性（conservation），或是蛋白质结构域的三级结构与二级结构，甚至是个别的氨基酸或核苷酸。

二、当前研究的成果

1. 软件

（1）Blast

在生物信息学，BLAST（基本的局部比对搜索工具）是一种算法，用于比较主生物的序列信息，诸如氨基酸序列的蛋白质的或核苷酸的 DNA 和/或 RNA 序列。BLAST 搜索使研究人员能够将查询序列与序列的库或数据库进行比较，并识别类似于高于特定阈值的查询序列的库序列。

根据查询序列可以获得不同类型的 BLAST。例如，在发现小鼠中先前未知的基因后，科学家通常会对人类基因组进行 BLAST 搜索，以查看人类是否携带相似的基因；BLAST 将基于序列的相似性鉴定人类基因组中与小鼠基因相似的序列。BLAST 算法和程序由美国国立卫生研究院的 Stephen Altschul，Warren Gish，Webb Miller，Eugene Myers 和 David J. Lipman

设计.BLAST 需要搜索序列以及要搜索的序列（也称为目标序列）或包含多个此类序列的序列数据库。BLAST 将在数据库中找到与查询中的子序列类似的子序列。

该软件网址：https://blast.ncbi.nlm.nih.gov/Blast.cgi

（2）CS-BLAST

与 BLAST 相比，使用 CS-BLAST 可将灵敏度提高一倍并显着提高对齐质量而不会降低速度。

（3）CUDASW ++

一种成熟的最先进的生物信息学软件，用于 Smith-Waterman 蛋白质数据库搜索，利用 NVIDIA GPU 的大规模并行 CUDA 架构，比 NCBI BLAST 快 10 到 50 倍进行序列搜索。

该软件网址：HTTP://cudasw.sourceforge.net/homepage.htm#latest

（4）DIAMOND

是一种新的高通量程序，用于将 DNA 读取或蛋白质序列与蛋白质参考数据库（如 NR）对齐，速度高达 BLAST 的 20,000 倍，具有高灵敏度。

该软件网址：http://ab.inf.uni-tuebingen.de/software/diamond/

2. 应用：

（1）序列组装

在生物信息学中，序列组装是指从较长的 DNA 序列比对和合并片段以重建原始序列。因为 DNA 测序技术无法一次性读取全基因组，而是根据所使用的技术读取 20 至 30000 个碱基的小片段。通常，称为读数的短片段来自鸟枪测序基因组 DNA 或基因转录物（EST）。将这些序列进行组装时需要运用序列比对的算法来提高组装的准确性。

（2）SNP 分析

其中来自不同个体的序列被比对以找到在群体中通常不同的单个碱基对。SNP 是我们对多种疾病易感性差异的基础（例如 - 镰状细胞性贫血，β-地中海贫血和由 SNP 引起的囊性纤维化）。疾病的严重程度和身体对治疗的反应方式也是遗传变异的表现。例如，APOE（载脂蛋白 E）基因中的单碱基突变与阿尔茨海默病的风险较低相关。

（3）将最佳匹配或对齐（OM）技术应用于社会科学序列数据

（4）通过多序列对齐引导词汇选择

（5）部分自动化比较方法语言学家传统上重建语言

（6）商业和营销研究应用多种序列比对技术来分析一系列购买时间

三、典型算法

（一）DNA 序列对比

1.动态规划 Dynamic programming

（一）通过 Needleman-Wunsch 算法产生全局比对

该算法由 Saul B. Needleman 和 Christian D. Wunsch 开发并于 1970 年出版，将一个大问题（例如完整序列）分成一系列较小的问题，并将解决方案用于较小的问题。重建解决更大问题的方法。它有时也被称为最优匹配算法和全局对齐技术。

算法：

（1）首先构建一个如上图 1 所示的网格。启动第三列顶部的第一个字符串，并在第三

行的开头处启动另一个字符串。填写列和行标题的其余部分。网格中应该没有数字。

		G	C	A	T	G	C	U
G								
A								
T								
T								
A								
C								
A								

（2）给每个字母打分

对应字母有三种情况：

Match：当前索引处的两个字母是相同的。

Mismatch：当前索引的两个字母不同。

Indel（INsertion 或 DELetion）：包含一个字母与另一个字符串中的间隙对齐。

将使用 Needleman 和 Wunsch 使用的系统：

匹配：+1

不匹配或不确定：-1

（3）填写表格

从第二行第二列的零开始。逐行移动单元格，计算每个单元格的分数。通过比较与单元格的左侧，顶部或左上角（对角线）相邻的单元格的得分并且为匹配、错配或插入添加适当的得分来计算得分。计算三种可能性中的每一种的候选分数：

来自顶部或左侧单元格的路径表示 indel 配对，因此请获取左侧和顶部单元格的分数，并将 indel 的分数添加到每个单元格中。

对角线路径表示匹配/不匹配，因此，如果行和列中的对应基数匹配，则取左上对角线单元格的分数并添加匹配分数，否则添加不匹配分数。

得到的单元格得分是三个候选得分中最高的。

鉴于第二行没有“顶部”或“左上”单元格，只有左侧的现有单元格可用于计算每个单元格的分数。因此，对于每个向右移位添加-1，因为这表示与先前分数不可擦除。这导致第一行为 0，-1，-2，-3，-4，-5，-6，-7。这同样适用于第二列，因为只能使用每个单元格上方的现有分数。因此得到的表是：

		G	C	A	T	G	C	U
	0	-1	-2	-3	-4	-5	-6	-7
G	-1							
A	-2							
T	-3							
T	-4							
A	-5							
C	-6							
A	-7							

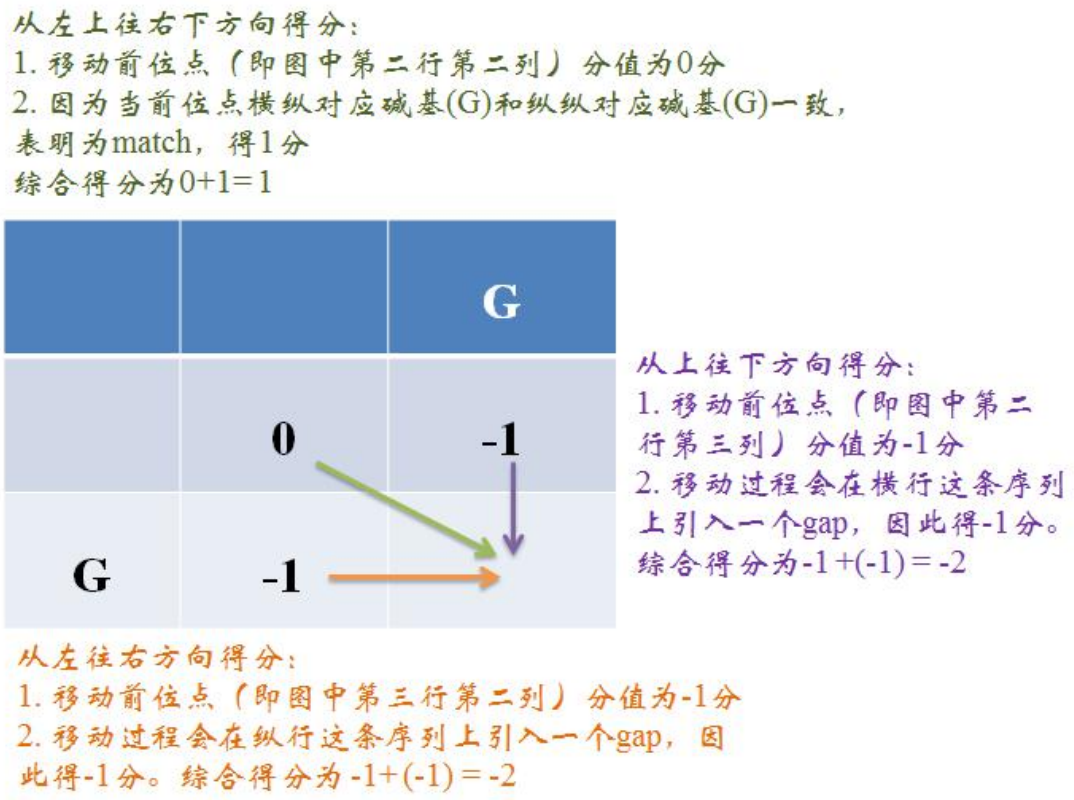
将其他单元格填满：

依次从左上往右下计算出每个位点的得分，计算时先算出从左，从上及从左上角移动到当前位点时的得分，这个得分值为：不同方向移动综合得分 = 移动前位点的得分 + 移动过程的得分

移动前位点的得分为移动前位点方框中的值，移动过程的得分按 1.1 中的得分约定计算如下：

从上往下和从左往右移动时都会引入 gap，前者是在横向这条序列上引入 gap，后者是在纵向这条序列上引入 gap，因此都会得-1 分；

从左上往右下方向移动时，如果当前位点横向和纵向对应碱基一致，表明为 match，得 1 分；如果当前位点横向和纵向对应碱基不一致，表明为 mismatch，得-1 分。



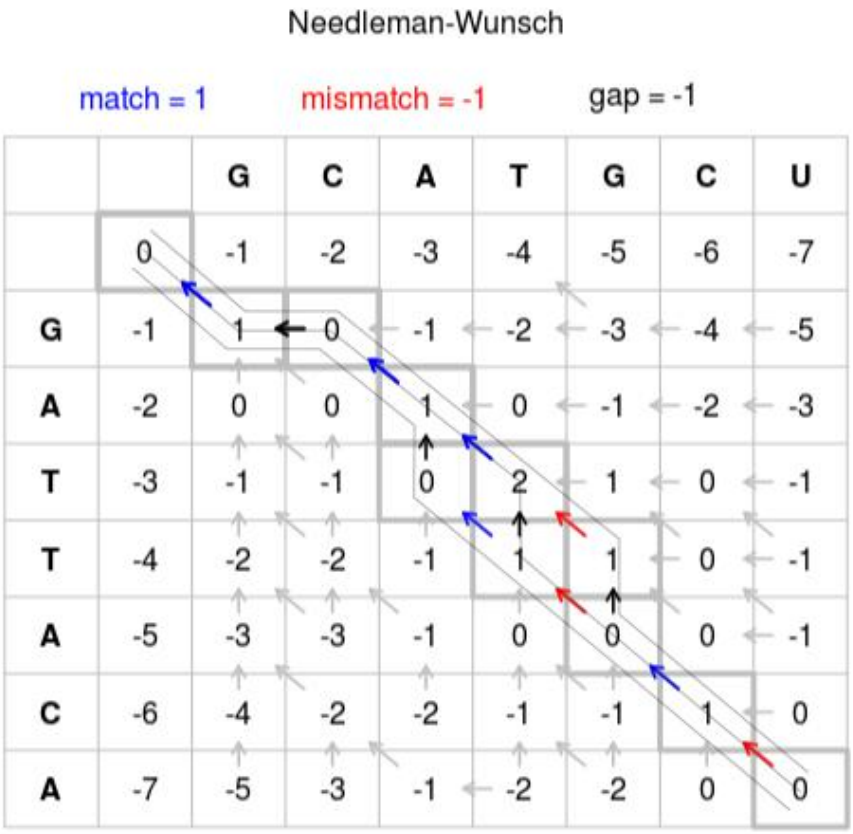
（4）跟踪箭头返回原点

按照箭头的方向标记从右下角的单元格返回到左上角的单元格的路径。从这条路径开始，序列由以下规则构成：

斜箭头表示匹配或不匹配，因此列的字母和原始单元格的行的字母将对齐。

水平或垂直箭头表示插入。水平箭头将间隙（“ - ”）与行的字母（“侧”序列）对齐，垂直箭头将间隙与列的字母对齐（“顶部”序列）。

如果有多个箭头可供选择，则它们代表对齐的分支。如果两个或多个分支都属于从左下角到右上角的单元格的路径，则它们是同样可行的对齐。在这种情况下，请将路径记录为单独的对齐候选。



序列最佳比对

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

U → CU → GCU → -GCU → T-GCU → AT-GCU → CAT-GCU → GCAT-GCU

A → CA → ACA → TACA → TTACA → ATTACA → -ATTACA → G-ATTACA

（2）通过 Smith-Waterman 算法进行局部比对

该算法用于确定两串核酸序列或蛋白质序列之间的相似区域，而不是着眼于整个序列。Smith-Waterman 算法比较所有可能长度的段和优化所述相似性度量。

该算法最初由 Temple F. Smith 和 Michael S. Waterman 于 1981 年提出。与 Needleman-Wunsch 算法一样，Smith-Waterman 是一种动态编程算法。因此，它保证找到关于所使用的评分系统的最佳局部对齐（其包括替换矩阵和间隙评分方案）。与

Needleman-Wunsch 算法的主要区别将负评分矩阵单元设置为零。回溯程序从最高得分矩阵单元开始并继续进行，直到遇到得分为零的单元格，产生得分最高的局部对齐。由于其在时间和空间上的二次复杂性，它通常不能实际应用于大规模问题，而是被替换为较不普遍但计算上更有效的替代方案，如（Gotoh，1982），（Altschul 和 Erickson， 1986），和（Myers and Miller，1988）。

算法：

（1）确定替代矩阵和差距惩罚方案。

替代矩阵 S（a,b）：构成两个序列的元素的相似度得分：每个碱基取代或氨基酸取代分配一个分数。通常，匹配被分配正分数，并且不匹配被分配相对较低的分数。以 DNA 序列为例。如果匹配得+1，不匹配得-1。

差距惩罚 Wk：有长度差距的罚款。一个简单的差距惩罚策略是对每个差距使用固定分数。然而，在生物学中，由于实际原因，需要对得分进行不同的计算。一方面，两个序列之间的部分相似性是常见现象; 另一方面，单个基因突变事件可导致插入单个长间隙。因此，形成长间隙的连接间隙通常比多个分散的短间隙更有利。为了考虑到这种差异，在评分系统中增加了间隙开放和间隙延伸的概念。差距开放分数通常高于差距延长分数。例如，EMBOSS Water 中的默认参数是：gap opening = 10，gap extension = 0.5。

（3）构建评分矩阵 H 并初始化其第一行和第一列。评分矩阵的大小是 （n + 1）*（m + 1）。注意基于 0 的索引

$$H_{k0} = H_{0l} = 0 \quad for \quad 0 \le k \le n \quad and \quad 0 \le l \le m$$

（4）使用下面的等式填充评分矩阵

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

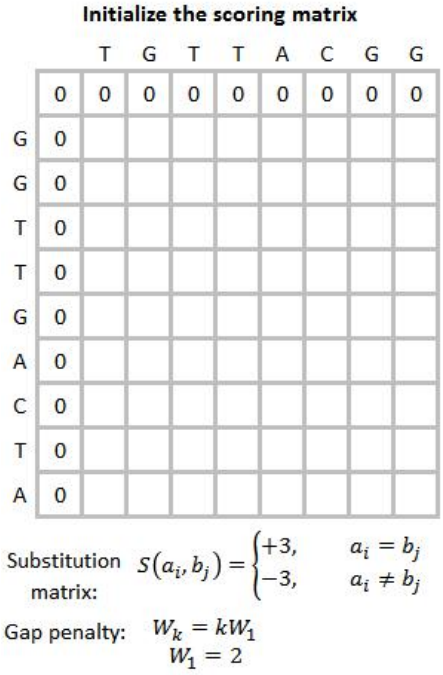
$H_{i-1,j-1} + s(a_i, b_j)$ 是对齐的得分 a_i 和 b_j，

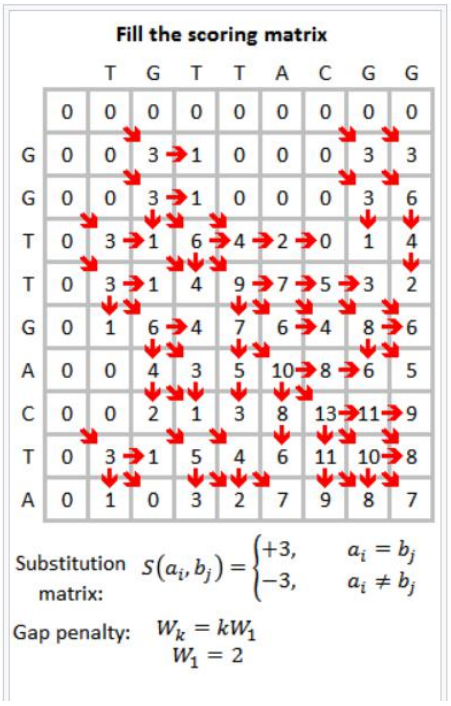
$H_{i-k,j} - W_k$ 是得分 a_i 是在长度差距的最后 k，

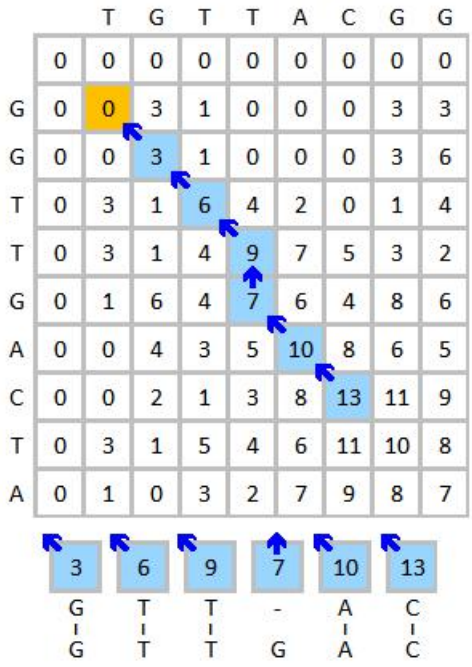
$H_{i,j-l} - W_l$ 是得分 b_j 是在长度差距的最后 l，

0 意味着没有相似之处 a_i 和 b_j。

（5）追溯。从评分矩阵中的最高分开始并且以得分为 0 的矩阵单元结束，递归地基于每个得分的来源追溯以生成最佳局部对齐。







from Wikipedia

2.单词方法

Word 方法，也称为 k 元组方法，是不能保证找到最佳对齐解决方案的启发式方法，但是比动态编程明显更有效。这些方法在大规模数据库搜索中特别有用，Word 方法以其在数据库搜索工具 FASTA 和 BLAST 系列中的实现而闻名。Word 方法在查询序列中识别一系列短的，非重叠的子序列（“Word”），然后与候选数据库序列匹配。减去被比较的两个序列中的单词的相对位置以获得偏移；如果多个不同的单词产生相同的偏移，这将指示对齐区域。只有在检测到该区域时，这些方法才适用更灵敏的对齐标准；因此，消除了与没有明显相似性的序列的许多不必要的比较。

在 FASTA 方法中，用户定义值 k 以用作搜索数据库的字长。该方法较慢但在较低的 k 值下更敏感，这对于涉及非常短的查询序列的搜索也是优选的。BLAST 搜索方法系列提供了许多针对特定类型的查询优化的算法，例如搜索远距离相关的序列匹配。开发 BLAST 是为了更快地替代 FASTA，而不会牺牲很多精度; 像 FASTA 一样，BLAST 使用长度为 k 的单词搜索，但仅评估最重要的单词匹配，而不是像 FASTA 那样评估每个单词匹配。大多数 BLAST 实现使用针对查询和数据库类型优化的固定默认字长，并且仅在特殊情况下更改，例如在使用重复或非常短的查询序列进行搜索时。

3. 点阵方法

点阵方法隐含地为单个序列区域产生一系列比对，虽然耗时大规模分析，但在定性和概念上都很简单。在没有噪声的情况下，可以很容易地从点阵图中可视地识别某些序列特征 - 例如插入，缺失，重复或反向重复。为了构建点阵图，两个序列沿着二维矩阵的顶行和最左列写入，点放置在适当列中的字符匹配的任何点 - 这是典型的递归图。一些实现根据两个字符的相似程度改变点的大小或强度，以适应保守替换。非常密切相关的序列的点图将沿着矩阵的主对角线显示为单线。

（二）多序列比对

多序列比对是成对比对的扩展，一次包含两个以上的序列。多个对齐方法尝试对齐给定查询集中的所有序列。多重比对通常用于鉴定假设为进化相关的一组序列上的保守序列区域。此类保守的序列基序可以结合使用具有结构和机械信息来定位该催化活性位点的酶。通过构建系统发育树，对齐也用于帮助建立进化关系。多序列比对在计算上难以产生，并且大多数问题的表达导致 NP 完全组合优化问题。]然而，这些比对在生物信息学中的实用性已导致开发出适合于比对三种或更多种序列的多种方法。

1.动态编程

动态规划技术理论上适用于任何数量的序列; 然而，因为它在时间和记忆方面都是计算上昂贵的，所以它很少用于最基本形式的三个或四个序列。该方法需要构建由两个序列形成的序列矩阵的 n 维等价物，其中 n 是查询中的序列数。首先在所有查询序列对上使用标准动态编程，然后通过考虑中间位置处的可能匹配或间隙来填充“对齐空间”，最终在每个两序列比对之间构建对齐。尽管该技术在计算上是昂贵的，但是在仅需要精确对准少数序列的情况下，其对全局最优解的保证是有用的。在 MSA 软件包中已经实现了一种用于降低动态编程的计算需求的方法，其依赖于“对的总和”目标函数。

2.渐进方法

渐进式，分层式或树型方法通过首先对齐最相似的序列，然后将相继较少的相关序列或组添加到比对中来生成多序列比对，直到整个查询集合已合并到解决方案中。描述序列相关性的初始树基于成对比较，可能包括类似于 FASTA 的启发式成对比对方法。渐进比对结果取决于“最相关”序列的选择，因此可能对初始成对比对中的不准确性敏感。大多数渐进多序列比对方法还根据它们的相关性对查询集中的序列进行加权，这降低了对初始序列进行较差选择的可能性，从而提高了比对精度。

Clustal 渐进式实施的许多变体用于多序列比对，系统发育树构建，以及作为蛋白质结构预测的输入。渐进方法的较慢但更准确的变体被称为 T-Coffee。

3.迭代方法

迭代方法试图改善对初始成对比对的准确性的严重依赖性，这是渐进方法的弱点。迭代方法通过分配初始全局对齐然后重新排列序列子集，基于所选择的对齐评分方法来优化目标函数。然后，重新排列的子集本身对齐以产生下一次迭代的多序列比对。

4.主题发现

基序发现（也称为概况分析）构建全局多序列比对，其试图在查询集中的序列之间比对短保守序列基序。这通常通过首先构建一般的全局多序列比对来完成，之后高度保守的区域被分离并用于构建一组轮廓矩阵。每个保守区域的分布矩阵排列成评分矩阵，但其每个位置的每个氨基酸或核苷酸的频率计数来自保守区域的特征分布，而不是来自更一般的经验分布。然后使用轮廓矩阵来搜索其他序列以寻找它们表征的基序的出现。在原来的情况下数据集包含少量序列，或仅包含高度相关的序列，添加伪拷贝以标准化主题中表示的字符分布。

5.受计算机科学启发的技术

计算机科学中常用的各种通用优化算法也已应用于多序列比对问题。隐马尔可夫模型已被用于为给定查询集的一系列可能的多序列比对产生概率分数; 尽管早期基于 HMM 的方法产生了令人沮丧的性能，但后来的应用已经发现它们在检测远程相关序列方面特别有效，因为它们不易受保守或半保守取代产生的噪声的影响。遗传算法和模拟退火也可用于优化多序列比对得分，如通过像对和方法这样的评分函数判断。更多完整的细节和软件包可以在主要文章多序列比对中找到。

该 Burrows-Wheeler 变换已成功应用于快速流行的工具，如短读对齐蝴蝶结和 BWA。请参阅 FM-index。

四、算法实现与测试

```
/*
 needleman_wunsch.c

 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "needleman_wunsch.h"

nw_aligner_t* needleman_wunsch_new()
{
   nw_aligner_t *nw = calloc(1, sizeof(nw_aligner_t));
   return nw;
}

void needleman_wunsch_free(nw_aligner_t *nw)
{
   aligner_destroy(nw);
   free(nw);
}

void needleman_wunsch_align(const char *a, const char *b,
```

```
                                        const scoring_t *scoring,
                                        nw_aligner_t *nw, alignment_t *result)
{
  needleman_wunsch_align2(a, b, strlen(a), strlen(b), scoring, nw, result);
}

void needleman_wunsch_align2(const char *a, const char *b,
                                         size_t len_a, size_t len_b,
                                         const scoring_t *scoring,
                                         nw_aligner_t *nw, alignment_t *result)
{
  aligner_align(nw, a, b, len_a, len_b, scoring, 0);

  // work backwards re-tracing optimal alignment, then shift sequences into place

  // note: longest_alignment = strlen(seq_a) + strlen(seq_b)
  size_t longest_alignment = nw->score_width-1 + nw->score_height-1;
  alignment_ensure_capacity(result, longest_alignment);

  // Position of next alignment character in buffer (working backwards)
  size_t next_char = longest_alignment-1;

  size_t arr_size = nw->score_width * nw->score_height;

  // Get max score (and therefore current matrix)
  enum Matrix curr_matrix = MATCH;
  score_t curr_score = nw->match_scores[arr_size-1];

  if(nw->gap_b_scores[arr_size-1] >= curr_score)
  {
    curr_matrix = GAP_B;
    curr_score = nw->gap_b_scores[arr_size-1];
  }

  if(nw->gap_a_scores[arr_size-1] >= curr_score)
  {
    curr_matrix = GAP_A;
    curr_score = nw->gap_a_scores[arr_size-1];
  }

  #ifdef SEQ_ALIGN_VERBOSE
    alignment_print_matrices(nw);
  #endif
```

```
result->score = curr_score;
char *alignment_a = result->result_a, *alignment_b = result->result_b;

// coords in score matrices
size_t score_x = nw->score_width-1, score_y = nw->score_height-1;
size_t arr_index = arr_size - 1;

for(; score_x > 0 && score_y > 0; next_char--)
{
  #ifdef SEQ_ALIGN_VERBOSE
  printf("matrix: %s (%lu,%lu) score: %i\n",
         MATRIX_NAME(curr_matrix), score_x-1, score_y-1, curr_score);
  #endif

  switch(curr_matrix)
  {
    case MATCH:
      alignment_a[next_char] = nw->seq_a[score_x-1];
      alignment_b[next_char] = nw->seq_b[score_y-1];
      break;

    case GAP_A:
      alignment_a[next_char] = '-';
      alignment_b[next_char] = nw->seq_b[score_y-1];
      break;

    case GAP_B:
      alignment_a[next_char] = nw->seq_a[score_x-1];
      alignment_b[next_char] = '-';
      break;

    default:
      fprintf(stderr, "Program error: invalid matrix number\n");
          exit(EXIT_FAILURE);
  }

  if(score_x > 0 && score_y > 0)
  {
    alignment_reverse_move(&curr_matrix, &curr_score,
                           &score_x, &score_y, &arr_index, nw);
  }
}

// Gap in A
```

```
    while(score_y > 0)
    {
        alignment_a[next_char] = '-';
        alignment_b[next_char] = nw->seq_b[score_y-1];
        next_char--;
        score_y--;
    }

    // Gap in B
    while(score_x > 0)
    {
        alignment_a[next_char] = nw->seq_a[score_x-1];
        alignment_b[next_char] = '-';
        next_char--;
        score_x--;
    }

    // Shift alignment strings back into 0th position in char arrays
    int first_char = next_char+1;
    int alignment_len = longest_alignment - first_char;

    // Use memmove
    memmove(alignment_a, alignment_a+first_char, alignment_len);
    memmove(alignment_b, alignment_b+first_char, alignment_len);

    alignment_a[alignment_len] = '\0';
    alignment_b[alignment_len] = '\0';

    result->length = alignment_len;
}


/*
  smith_waterman.c
  */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "sort_r/sort_r.h"

#include "smith_waterman.h"
#include "alignment_macros.h"
```

```
typedef struct {
    uint32_t *b; size_t l, s; // l is bits, s in uint32_t
} BitSet;

static inline BitSet* bitset_alloc(BitSet *bs, size_t l) {
    bs->s = (l+31)/32;
    bs->l = l;
    bs->b = calloc(sizeof(bs->b[0]), bs->s);
    return bs->b ? bs : NULL;
}

static inline void bitset_dealloc(BitSet *bs) {
    free(bs->b);
    memset(bs, 0, sizeof(*bs));
}

static inline BitSet* bitset_set_length(BitSet *bs, size_t l) {
    size_t ss = (l+31)/32;
    if(ss > bs->s) {
        bs->b = realloc(bs->b, ss*sizeof(bs->b[0]));
        memset(bs->b+bs->s, 0, (ss-bs->s)*sizeof(bs->b[0])); // zero new memory
        bs->s = ss;
    }
    bs->l = l;
    return bs->b ? bs : NULL;
}

// For iterating through local alignments
typedef struct
{
    BitSet match_scores_mask;
    size_t *sorted_match_indices, hits_capacity, num_of_hits, next_hit;
} sw_history_t;

// Store alignment here
struct sw_aligner_t
{
    aligner_t aligner;
    sw_history_t history;
};

// Sort indices by their matrix values
// Struct used to pass data to sort_match_indices
```

```
typedef struct
{
   score_t *match_scores;
   unsigned int score_width;
} MatrixSort;

// Function passed to sort_r
int sort_match_indices(const void *aa, const void *bb, void *arg)
{
   const size_t *a = aa;
   const size_t *b = bb;
   const MatrixSort *tmp = arg;

   // Recover variables from the struct
   const score_t *match_scores = tmp->match_scores;
   size_t score_width = tmp->score_width;

   long diff = (long)match_scores[*b] - match_scores[*a];

   // Sort by position (from left to right) on seq_a
   if(diff == 0) return (*a % score_width) - (*b % score_width);
   else return diff > 0 ? 1 : -1;
}

static void _init_history(sw_history_t *hist)
{
   bitset_alloc(&hist->match_scores_mask, 256);
   hist->hits_capacity = 256;
   size_t mem = hist->hits_capacity * sizeof(*(hist->sorted_match_indices));
   hist->sorted_match_indices = malloc(mem);
}

static void _ensure_history_capacity(sw_history_t *hist, size_t arr_size)
{
   if(arr_size > hist->hits_capacity) {
      hist->hits_capacity = ROUNDUP2POW(arr_size);
      bitset_set_length(&hist->match_scores_mask, hist->hits_capacity);

      size_t mem = hist->hits_capacity * sizeof(*(hist->sorted_match_indices));
      hist->sorted_match_indices = realloc(hist->sorted_match_indices, mem);
      if(!hist->match_scores_mask.b || !hist->sorted_match_indices) {
         fprintf(stderr, "%s:%i: Out of memory\n", __FILE__, __LINE__);
         exit(EXIT_FAILURE);
      }
```

```
    }
}

sw_aligner_t* smith_waterman_new()
{
    sw_aligner_t *sw = calloc(1, sizeof(sw_aligner_t));
    _init_history(&(sw->history));
    return sw;
}

void smith_waterman_free(sw_aligner_t *sw)
{
    aligner_destroy(&(sw->aligner));
    bitset_dealloc(&sw->history.match_scores_mask);
    free(sw->history.sorted_match_indices);
    free(sw);
}

aligner_t* smith_waterman_get_aligner(sw_aligner_t *sw)
{
    return &sw->aligner;
}

void smith_waterman_align(const char *a, const char *b,
                                    const scoring_t *scoring, sw_aligner_t *sw)
{
    smith_waterman_align2(a, b, strlen(a), strlen(b), scoring, sw);
}

void smith_waterman_align2(const char *a, const char *b,
                                     size_t len_a, size_t len_b,
                                     const scoring_t *scoring, sw_aligner_t *sw)
{
    aligner_t *aligner = &sw->aligner;
    sw_history_t *hist = &sw->history;
    aligner_align(aligner, a, b, len_a, len_b, scoring, 1);

    size_t arr_size = aligner->score_width * aligner->score_height;
    _ensure_history_capacity(hist, arr_size);

    // Set number of hits
    memset(hist->match_scores_mask.b, 0, (hist->match_scores_mask.l+31)/32);
    hist->num_of_hits = hist->next_hit = 0;
```

```
    size_t pos;
    for(pos = 0; pos < arr_size; pos++) {
        if(aligner->match_scores[pos] > 0)
            hist->sorted_match_indices[hist->num_of_hits++] = pos;
    }

    // Now sort matched hits
    MatrixSort tmp_struct = {aligner->match_scores, aligner->score_width};
    sort_r(hist->sorted_match_indices, hist->num_of_hits,
                sizeof(size_t), sort_match_indices, &tmp_struct);
}

// Return 1 if alignment was found, 0 otherwise
static char _follow_hit(sw_aligner_t* sw, size_t arr_index,
                                        alignment_t* result)
{
    const aligner_t *aligner = &(sw->aligner);
    const sw_history_t *hist = &(sw->history);

    // Follow path through matrix
    size_t score_x = (size_t)ARR_2D_X(arr_index, aligner->score_width);
    size_t score_y = (size_t)ARR_2D_Y(arr_index, aligner->score_width);

    // Local alignments always (start and) end with a match
    enum Matrix curr_matrix = MATCH;
    score_t curr_score = aligner->match_scores[arr_index];

    // Store end arr_index and (x,y) coords for later
    size_t end_arr_index = arr_index;
    size_t end_score_x = score_x;
    size_t end_score_y = score_y;
    score_t end_score = curr_score;

    size_t length;

    for(length = 0; ; length++)
    {
        if(bitset32_get(hist->match_scores_mask.b, arr_index)) return 0;
        bitset32_set(hist->match_scores_mask.b, arr_index);

        if(curr_score == 0) break;

        //printf(" %i (%i, %i)\n", length, score_x, score_y);
```

```
    // Find out where to go next
    alignment_reverse_move(&curr_matrix, &curr_score,
                                        &score_x, &score_y, &arr_index, aligner);
}

// We got a result!
// Allocate memory for the result
result->length = length;

alignment_ensure_capacity(result, length);

// Jump back to the end of the alignment
arr_index = end_arr_index;
score_x = end_score_x;
score_y = end_score_y;
curr_matrix = MATCH;
curr_score = end_score;

// Now follow backwards again to create alignment!
unsigned int i;

for(i = length-1; curr_score > 0; i--)
{
    switch(curr_matrix)
    {
        case MATCH:
            result->result_a[i] = aligner->seq_a[score_x-1];
            result->result_b[i] = aligner->seq_b[score_y-1];
            break;

        case GAP_A:
            result->result_a[i] = '-';
            result->result_b[i] = aligner->seq_b[score_y-1];
            break;

        case GAP_B:
            result->result_a[i] = aligner->seq_a[score_x-1];
            result->result_b[i] = '-';
            break;

        default:
        fprintf(stderr, "Program error: invalid matrix in _follow_hit()\n");
             exit(EXIT_FAILURE);
    }
```

```
        alignment_reverse_move(&curr_matrix, &curr_score,
                                        &score_x, &score_y, &arr_index, aligner);
    }

    result->result_a[length] = '\0';
    result->result_b[length] = '\0';

    result->score = end_score;

    result->pos_a = score_x;
    result->pos_b = score_y;

    result->len_a = end_score_x - score_x;
    result->len_b = end_score_y - score_y;

    return 1;
}

int smith_waterman_fetch(sw_aligner_t *sw, alignment_t *result)
{
    sw_history_t *hist = &(sw->history);

    while(hist->next_hit < hist->num_of_hits)
    {
        size_t arr_index = hist->sorted_match_indices[hist->next_hit++];
        // printf("hit %lu/%lu\n", hist->next_hit, hist->num_of_hits);

        if(!bitset32_get(hist->match_scores_mask.b, arr_index) &&
             _follow_hit(sw, arr_index, result))
        {
            return 1;
        }
    }

    return 0;
}
```

输出：

```
➜ bin git:(master) ./needleman_wunsch AGTGCTGAAGTTCGCCAGTTGACG AGTGCTGAAAGTTGCGCCAGTGAC
AGTGCTGAA-GTT-CGCCAGTTGACG
AGTGCTGAAAGTTGCGCCAGT-GAC-
```

```
➜ bin git:(master) ./smith_waterman AGTGCTGAAAGTTGCGCCAGTGAC AGTGCTGAAGTTCGCCAGTTGACG
== Alignment 0 lengths (24, 24):
hit 0.0 score: 35
  AGTGCTGAAAGTTGCGCCAGT-GAC  [pos: 0; len: 24]
  AGTGCTGAA-GTT-CGCCAGTTGAC  [pos: 0; len: 23]
```

五、性能评价

1. Needleman-Wunsch 算法

计算得分 F{ij}对于表中的每个单元格是一个 O（1）操作，因此，算法的时间复杂度对于两个长度分别为 n 和 m 的序列是 O(MN)。已经表明可以通过使用 Method of Four Russians（能够加速涉及布尔矩阵的算法），改善运行时间为 O（mn / log n）。因为算法填写了 m*n 的表，所以空间的复杂性是 O（MN）。

2. Smith-Waterman 算法

Smith-Waterman 算法对时间要求很高：要对齐两个长度为 m 和 n 的序列，所以 O(m n)的时间复杂度是必需的。Gotoh 和 Altschul 将算法优化为 O（MN）。空间复杂度是由 Myers 和 Miller 优化从 O（MN）至 O（n）（线性），n 为短序列的长度。

3. 比较

全局性配对主要应用于比较同源性的基因的相似性，而局部性配对主要应用于在非同源性的基因序列中寻找具有相似性的同源的基因域。

局部性配对

```
5' ACTACTAGATTACTTACGGATCAGGTACTTTAGAGGCTTGCAACCA 3'
             |||| |||||| ||||||||||||||||
         5' TACTCACGGATGAGGTACTTTAGAGGC 3'
```

全局性配对

```
5' ACTACTAGATTACTTACGGATCAGGTACTTTAGAGGCTTGCAACCA 3'
   ||||||||||    |||||||  |||||||||||||| |||||||
5' ACTACTAGATT----ACGGATC--GTACTTTAGAGGCTAGCAACCA 3'
```

两种算法都使用了置换矩阵，空位罚分，得分矩阵，以及回溯的方法，具有一定的相似度。然而，史密斯-沃特曼算法与尼德曼-翁施算法的三个主要区别在于：

	史密斯-沃特曼算法	尼德曼-翁施算法
初始化阶段	第一行和第一列全填充为 0	首行和首列需要考虑空位罚分
打分阶段	若得分为负，则分数为零	得分可以为负
回溯阶段	从最高分开始，回溯直至得分为 0	从右下角开始，回溯至左上角

两个算法时间复杂度相差不大，但是对于长度相差较多的两个序列，即使有很多能够对齐的部分，Needleman-Wunsch 算法匹配程度将会很低。因为该配对方法要求将两个序列的所有碱基都进行配对，所以当两序列长度相差很大时，将会造成配对结果含有很长的间隔。尽管在某处的配对序列和参考序列的匹配程度相当高，在全局性配对时，不会把这两个片段直接进行配对，而是尽量尝试让参考序列内的所有碱基都参与匹配，造成结果与局部性配对方法有极大差别。

因此，在选择算法时要考虑长度差、是否同源等因素。

4. 优化

快速优化整体序列比对算法（FOGSAA），相比中的 Needleman-Wunsch 算法时，FOGSAA 达到 70-90％为高度相似核苷酸序列的时间增益（具有>80％的相似性），并且对于具有 30-80％的相似性的序列 54-70％。

https://www.researchgate.net/publication/236459598_FOGSAA_Fast_optimal_global_sequence_alignment_algorithm

https://www.nature.com/articles/srep01746

六、研究结论

1. 对于动态规划的理解：

特点：

（1）序列对比两种算法的解都能用递归关系表示。

（2）用递归方法对这些递归关系的直接实现会造成解决方案效率低下，因为其中包含了对子问题的多次计算。

（3）一个问题的最优解能够用原始问题的子问题的最优解构造得到。

2. 动态编程可能是计算机科学在生物学上最重要的应用，但不是唯一的应用。生物信息学和计算生物学是交叉学科领域，正在迅速成为具有专门的学术程序的独立学科。

其他常用算法有：基本字符串算法、后缀树算法、非精确匹配算法、映射与测序。通过对 DNA 序列、蛋白质序列的分析，可以构建遗传图谱（genetic maps）、物理图谱（physical maps），应用于"cDNA 战略"、辐射性杂交制图技术（radiation-hybrid mapping）等。

此外，图论、概率论等内容在生物信息学中均有广泛的使用。