文章编号:1672－2477(2008)04－0052－05

利用矩阵搜索求所有最长公共子序列的算法

宫洁卿

(东南大学 软件学院,江苏 南京　211189)

摘要:利用动态规划法求出二维数组的情况下,使用矩阵搜索的方法求出所有分支,从而求出所有最长公共子序列的算法.该算法将通常认为的指数量级的时间复杂度降低到了 $\max\{O(cmn),O(ck)\}$.随后对此算法的正确性以及效率做了证明.

关　键　词:最长公共子序列(LCS);矩阵搜索;算法

中图分类号:TP301.6　　**文献标识码**:A

前　言

最长公共子序列(Longest Common Subsequence)算法是一种非常基础的算法.其主要目的是找出两个序列中最长的公共子序列.目前 LCS 算法在生物工程上有着广泛的应用.

在生物工程中,基因序列是保存生物基本信息的地方,包含了海量的生物数据,基因由 4 种不同的碱基通过不同的组合来表现,人类基因有将近 30 亿个 DNA 碱基对,呈双螺旋结构.随着基因序列长度的增长,其组合的数量也是以指数级增长,通过人工对 DNA 的差异进行比较分析是难以实现的.因此,分子生物学家越来越依靠高效的计算机字符串比较算法,即将基因序列的问题转换为由 4 种基本字符组合出来的字符串,并以此为基础进行处理.在基因工程中,经常需要比较位于相同位置的两条不同的基因序列,根据找出的公共自序列来对样本进行分析.在常见的基因分析中,一般需要找到所有的最长公共子序列,此时 LCS 算法成为首选.然而,目前对 LCS 算法的研究一般针对的是对某一个最长公共子序列的算法进行优化和分析,对求出所有的最长公共子序列的研究,目前尚无针对此问题的优秀算法[2~5].传统的算法都是基于路径依赖的关系求所有公共自序列.本文利用动态规划法,在计算出二维数组的情况下,提出一种称为矩阵搜索的方法求出所有分支,从而求出所有最长公共子序列的算法,该算法将通常认为的指数量级的时间复杂度降低到了 $\max\{O(cmn),O(ck)\}$.

1　背景知识

1.1　LCS 算法的定义[1]

定义 1　子序列的概念:设 $X=(x_1,x_2,\cdots,x_m)$,若有 $1\leqslant i_1<i_2<\cdots<i_k\leqslant m$,使得 $Z=(z_1,z_2,\cdots,z_k)=(x_{i1},x_{i2},\cdots,x_{ik})$,则称 Z 是 X 的子序列,记为 $Z<X$.

e. g.　$X=(A,B,C,B,D,A,B)$, $Z=(B,C,B,A)$, 则有 $Z<X$.

定义 2　公共子序列的概念:设 X,Y 是两个序列,且有 $Z<X$ 和 $Z<Y$, 则称 Z 是 X 和 Y 的公共子序列.

定义 3　最长公共子序列的概念:若 $Z<X,Z<Y$,且不存在比 Z 更长的 X 和 Y 的公共子序列,则称 Z 是 X 和 Y 的最长公共子序列,记为 $Z\in \mathrm{LCS}(X,Y)$. 最长公共子序列往往不止一个.

e. g.　$X=(A,B,C,B,D,A,B)$, $Y=(B,D,C,A,B,A)$, 则 $Z=(B,C,B,A)$, $Z'=(B,C,A,B)$, $Z''=(B,D,A,B)$ 均属于 $\mathrm{LCS}(X,Y)$,即 X,Y 有 3 个 LCS.

1.2　传统解法

传统的 LCS 解法是采用动态规划算法(Dynamic Programing)来解决.其解法可以用一个递归函数来表示:

收稿日期:2008－08－10

作者简介:宫洁卿(1984—),男,安徽芜湖人,硕士研究生.

$$C[i,j]=\left\{\begin{array}{l}0 \text{ 若 } i=0 \text{ 或 } j=0\\ C[i-1,j-1]+1 \text{ 若 } i,j>0 \text{ 且 } x_i=y_j\\ \max\{C[i-1,j],C[i,j-1]\} \text{ 若 } i,j>0 \text{ 且 } x_i\neq y_i\end{array}\right\}$$

引进一个二维数组 C,用 $C[i,j]$ 记录 X_i 与 Y_j 的 LCS 的长度. 如果是自底向上进行递推计算,那么在计算 $C[i,j]$ 之前,$C[i-1,j-1]$, $C[i-1,j]$ 与 $C[i,j-1]$ 均已计算出来. 此时,根据 $X[i]=Y[j]$ 还是 $X[i]\neq Y[j]$,就可以计算出 $C[i,j]$. 每一个 $C[I,j]$ 还负责纪录自己是通过哪个子问题的值求得的. 一共有 5 种可能:从左边,从上边,从左上,从左边和上边,无.

2　基于矩阵搜索的算法

2.1　二维数组 C 的分析

本算法是基于通过动态规划算法给出的二维数组 C 为基础进行计算的.

根据图 1 的矩阵可以看到,每一个从 3→2,2→1,1→0 的路线就是需要的结果. 如果是使用穷举的方法走所有的路线,那么获得的结果很多都是重复的,并且比较是否是不同的结果也会非常的麻烦.

通过观察图 1 可以发现,如果可以从某个 $(n+1)\rightarrow n$ 出发,能够用某种手段得到所有的 $n\rightarrow(n-1)$,那么其中 $n\rightarrow n$ 的路线并不需要去关心,只要证明从 $(n+1)\rightarrow n$ 到 $n\rightarrow(n-1)$ 是连通的就可以.

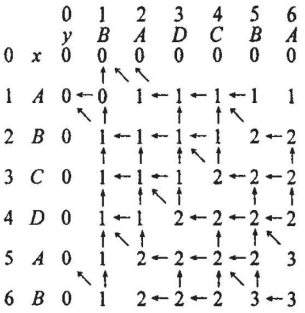

图 1　二维矩阵一

2.2　传统解法以及缺点

以图 1 为例,传统的解法是从二维矩阵的右下角考虑搜索所有的路线. 这种方法最大的缺点就是要搜索所有的路径. 在最坏的情况下,搜索了 $O(C2^n)$ 数量级的路线,只得到 1 个结果.

目前现有的优化方法是避免走重复的路线. 算法将所走过的路径压入栈中,在走不同的路径时与栈中的元素比较,如果是曾经走过的路径且没有新的分支,则直接抛弃掉.

传统解法的最大缺点是在于:算法还是基于自顶向下的分支搜索,必须要走遍所有的分支,且在避免走重复路径的时候,不得不做大量的比较. 为了进行快速的比较,数据的存储也比较复杂.

2.3　矩阵搜索算法的基本思路和步骤

(1) 本算法有两个栈. 第一个栈 store 用于存放所有搜索到的元素,当该栈为空时,运算结束. 第二个栈 print 用于存放准备打印的元素. 先计算出二维数组 C,每个节点都纪录本节点所在的坐标,LCS 长度和指向的方向. 假设该 LCS 串最长为 n,那么虚拟一个 $(n+1)\rightarrow n$ 的节点,该节点的坐标位于数组 C 最右下角的节点 $C[i,j]$ 的右下方,为 $C[i+1,j+1]$. 将该节点压入栈 store.

(2) 检测栈 store 是否为空,如果为空,则本算法结束.

(3) 从 store 栈顶取出一个节点.

(4) 如果当前的元素是边界元素 1→0 时,将该元素压入栈 print,打印栈 print 里面除了栈底的所有元素(无须打印原本不存在的虚拟结点). 查看 store 栈里最上面一个元素的 LCS 长度. 弹出栈 *print* 里面所有 LCS 长度比 store 栈最上面的 LCS 长度大或相等的元素,跳转到第 2 步.

(5) 将第三步取出的结点压入 print 栈.

(6) 设该结点的 LCS 长度为 $n+1$,从该节点出发,查看该节点的方向箭头,按照斜方向路线、向上的路线(如果是双向的路线则走向上的路线)、向左的路线为优先级找到一个 $n\rightarrow(n-1)$ 的节点,假设为 $C[X_1,Y_1]$.

(7) 再次从该节点出发,查看该节点的方向箭头,按照斜方向路线、向左的路线(如果是双向的路线则走向左的路线)、向上的路线为优先级找到一个 $n\rightarrow(n-1)$ 的节点,假设为 $C[X_2,Y_2]$.

(8) 比较 $C[X_1,Y_1]$ 和 $C[X_2,Y_2]$ 这两个节点. 如果这两个结点是同一个结点,则将该结点压入栈 store,跳转到第 2 步.

(9) 从刚才得到的两个结点 $C[X_1,Y_1]$ 和 $C[X_2,Y_2]$,在二维数组 C 中,以 (X_1,Y_1)(X_2, Y_1)(X_1, Y_2)(X_2,Y_2) 这四个点为坐标所构成的矩阵,在该矩阵中搜索所有的元素,将所有满足 $n\rightarrow(n-1)$ 的节点

压入栈 store. 跳转到第 2 步.

2.4 实验结果

本算法使用了 4 个例子来测试性能:

使用例子 1:字符串 1:*ABCDCDABCD* 字符串 2:*BADCDCBADC*

使用例子 2:字符串 1:*ABCDCDABCDABCDCDABCD* 字符串 2:*BADCDCBADCBADCDCBADC*

使用例子 3:字符串 1:*ABCDCDABCDABCDCDABCDABCDCDABCD*

字符串 2:*BADCDCBADCBADCDCBADCBADCDCBADC*

使用例子 4:字符串 1:*ABCDCDABCDABCDCDABCDABCDCDABCDABCDCDABCD*

字符串 2:*BADCDCBADCBADCDCBADCBADCDCBADCBADCDCBADC*

得到的结果如表 1.

表 1 4 种字符串的运算结果

	字符串长度 /*bit*	结果数量 / 个	总比较次数 (矩阵搜索法) / 次	平均得出一个结果所比较的次数 / 次	指数量级比较的次数 / 次	平均得出一个结果所比较的次数(穷举法)/ 次
例子 1	10	20	311	15.55	$2^{10} = 1\,024$	51
例子 2	20	700	11 531	16.47	$2^{20} = 1\,048\,576$	1 498
例子 3	30	25 460	423 571	16.64	$2^{30} \approx 1.07 \times 10E9$	42 174
例子 4	40	936 540	1 5626 731	16.69	$2^{40} \approx 1.10 \times 10E12$	1 174 014

注 1:比较次数不包括进出栈的次数. 而且每得到一个结果进出栈的次数是常数量级的,不影响结果. 注 2:使用穷举法来进行运算是无法在有限的时间内得到结果的,因此以指数量级来代替比较的次数.

从以上的例子可以看出,随着结果数量变多,经过此算法的运算,平均每得出一个结果的比较次数固定在 16 到 17 次之间. 因此,比较次数相对于结果数量来说是线性的. 而利用穷举法运算的时候,平均得出一个结果所比较的次数是指数量级增长的.

这个差别的根本原因是矩阵搜索法不会有重复的结果,而利用穷举法得到的结果有很多重复的,重复的数量随着结果数量的增长而指数级的增长.

2.5 矩阵搜索算法的优势

该矩阵搜索算法最大的好处是,避免了依靠分支路线来判断是否找到目标线路的缺点. 传统解法中,无论怎样对分支进行搜索优化,只要是基于分支路线的,都无法达到很高的效率. 正是因为本算法解决了分支路线的依赖问题,算法才会高效.

2.6 存在的问题

本算法还有以下需要解决的问题:问题 1:这个算法所构造出来的矩阵是否可以包含所有所需要的分支?问题 2:这个算法是否会包含我们不需要的分支?问题 3:如果包含所有的分支,它们是否和起始搜索点是连通的?

在解决问题之前,先从图 1 中得出一个结论:

结论 1 所有的元素的 LCS 长度值都会大于等于左边的或上边的元素. 从 2.2 节的递归函数可以证明这个结论.

这三个问题可以用反证法证明.

看图 2,是否可以构造出图中的矩阵呢?在这个例子中,$C[2,5]$ 的位置,它的箭头必定是两个方向的. 如果要让这个箭头向左而不向上,那么在 $C[1,5]$ 的位置必定应该为 0. 但从结论 1 可以知道:LCS 矩阵图的特点使这种可能性不存在.

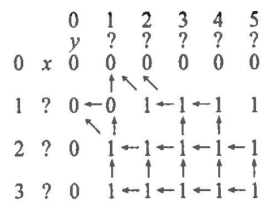

图 2 二维矩阵二

结论 2 从一个起点 $(n+1) \to n$ 开始搜索 $n \to (n-1)$ 的时候只有 3 种可能:找到;不断的向上前进,找到;向上前进到底,然后向左前进,找到. 这也就是算法第 6 步的步骤. 同理,第 7 步的搜索也是一样. 绝不可能出现图 2 的情况.

首先定义算法第 6、7 步搜索到的元素为边界元素. 那么当这个元素改变时,算法搜索的区域也会变

化. 在图 3 中 算法搜索的区域是 x 为 2 至 4、y 为 2 至 4 的区域. 假如 $C[5,3]$ 为 $1 \rightarrow 0$, $C[4,2]$ 就不再是边界元素了, 而 $C[5,3]$ 将成为边界元素. 同理 $C[5,4]$、$C[2,5]$、$C[3,5]$ 也都不会出现 $1 \rightarrow 0$. 否则, 这个搜索区域将会改变.

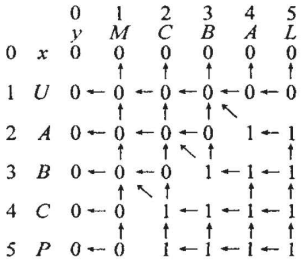

图 3　二维矩阵三

$C[4,5]C[5,4]C[5,5]$ 这三个元素所代表的区域也不可能出现 $1 \rightarrow 0$ 这样的边界元素, 否则将会违反结论 1. 同理 $C[1,1]C[1,2]C[1,3]C[2,1]C[3,1]$ 所代表的区域也不会出现边界元素.

$C[4,1]C[5,1]C[1,4]C[1,5]$ 所代表的区域因为结论 2 的原因, 不可能或不会出现我们所需要的元素.

推广到更一般的情况, 就得出结论 3.

结论 3　只需要搜索边界元素 $C[4,2]$ 和 $C[2,4]$ 所确定的结点矩阵就可以找到需要的所有结果.

结论 3 证明了问题 1.

既然从结论 3 得知, 所有的结果都包含在这个搜索矩阵内, 那么是否必然可以找到一个从起始点到目标点的路线呢?在图 4 中, $C[2,2]$ 是否有可能是 $2 \rightarrow 1$ 呢?假如 $C[2,2]$ 是 $2 \rightarrow 1$, 那么字符串 1 的第二个元素必定和字符串 2 的第二个元素相同, 则搜索矩阵就会发生改变. 因此, 从以上得出结论 4:

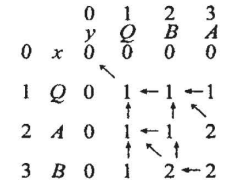

图 4　二维矩阵四

结论 4　只要搜索到目标元素, 必然存在从起始点到达该点的通路.

结论 3 和结论 4 回答了问题 2 和问题 3.

2.7　一般性结论

本矩阵搜索算法可以找到所有的目标分支元素, 不会遗漏或包含不需要的分支, 搜索到的分支必然存在着可以到达的通路.

3　算法的量级以及分析

考虑本算法量级的时候有三个影响因素:生成的结果数量 k;第一个字符串 x 的长度 m;第二个字符串 y 的长度 n. 因为从 3.3 的实验结果上我们可以看出, 长度为 40 的两个字符串可以生成将近一百万个结果, 那么就必须要考虑生成的结果数量 k. 从实验结果来看, 该算法的量级为 $O(ck)$. 那么, 是否会有例外的情况呢?图 6 是一个极端的例子, 算法搜索了 6×6 的矩阵, 只得到了两个解. 这时候, 该算法的量级就退化成 $O(cmn)$. 如果比较的字符串是 $ABCDAB$ 和 $BADCBA$, 能得到 10 个结果, 总比较次数为 111 次. 图 6 的例子只能得到 2 个结果, 总比较次数为 51 次.

从以上的例子得知:

结论 5　在字符串长度限定的情况下, 得到的结果越多, 比较次数就越多, 平均每得出一个结果的比较次数就越少.

通过实验和观察发现:3×3 的搜索矩阵数量越多, 限定长度的字符串产生结果数量越多. 如果一个二维数组 C 中的搜索矩阵过大或者过小, 那么产生的结果就会大大的减少. 因此, 虽然平均每得出一个 LCS 结果的比较次数的时间复杂度降到了 $O(cmn)$, 但总的比较次数却大大减少了.

那么在 3.3 的实验结果中, 为何平均每得出一个结果所花费的比较次数在 16 至 17 次之间呢?

图 1 可以规约成图 5 的形式. 当从 S(起点) 走到 0(终点) 时, 就得到了一个最长公共子序列. 走不同的路径就可以得到不同的结果.

实际上, 图 5 就是利用二维数组 C 得到最优化的图. 如利用图 5 求解所有 LCS, 那么这就是比较次数最少的最优解法.

从很多的实验中发现, 在限定长度的字符串下, 类似实验 3.3 的字符串所生成的结果数量是最多的. 随着字符串长度的增长, 生成的结果数量是指数级增长的. 将很长的字符串转化为二维数组时, 3×3 的矩阵是最多的. 该算法搜索一个 $3x3$ 的矩阵所花费的比较次数为:3(查找上斜箭头) + 3(查找左斜箭头) + 1(两个结点比较一次) + 9(搜索 $3x3$ 矩阵节点) = 16 次.

这个数字跟我们得到的平均比较次数是非常接近的. 而从类似图 5 的分析表明:本算法搜索 3×3 矩阵的数量是约等于结果数量的. 这也说明了为何随着比较结果的增长,平均每的到一个结果的比较次数恒定在 16 到 17 次之间的原因.

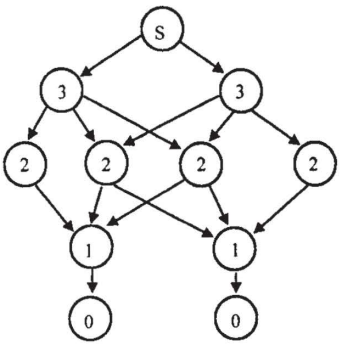

图 5 归约图

如果比较的字符串不像 3.3 所举的例子,用多种其它的组合,所得到的结果数量基本上会大幅度减少,每得出一个结果所花费的平均比较次数会大大增加. 其原因是搜索的矩阵'面积'变大了. 最终,时间复杂度会退化到 $O(cmn)$.

因此,在长度限定的字符串 m 和 n 中,如果生成的最长公共子序列数量 k_{min} 最少,则时间复杂度为 $O(cmn)$. 如果生成的最长公共子序列数量 k_{max} 最多,则时间复杂度为 $O(ck)$. 而如果生成的最长公共子序列数量 k 介于 k_{min} 和 k_{max} 之间,从上面的分析可得出:时间复杂度不会超过 $\max\{O(cmn),O(ck)\}$.

总结:从以上的分析得到该算法的时间复杂度为 $\max\{O(cmn),O(ck)\}$.

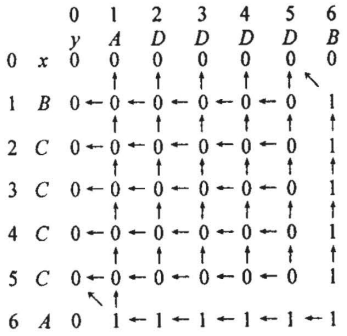

图 6 二维矩阵五

4 总 结

通过以上的分析以及证明,本算法可以很好的完成找出所有 LCS 的任务. 在前面的实验中发现,利用 AB,BA,CD,DC 这四种基本组合可以得到最多的 LCS. 在生物学中的基因碱基对排列中,基因的构造也是由这四个基本组合算子组合得来的. 该算法在处理生物学中基因排列这种需要极大运算量的工作是非常合适的. 如果只需要得出一个 LCS 结果,该算法就无法很好的完成任务. 本算法只适合需要求解所有 LCS 且结果数比较多的情况.

参考文献:

[1] 潘金贵. 现代计算机常用数据结构和算法[M]. 南京:南京大学出版社,1994,213 - 216.

[2] Hrischberg,D S. Algorithms for the Longest Common Subsequence Problem[J]. ACM,1997,24(4):664 - 675.

[3] Hirschberg,D S. A linear space algorithm for computing maximal common subsequences[J]. Commun. ACM, 18:341 - 343.

[4] A. Nandy. A new graphical representation and analysis of DNA sequence structure : I. Methodology and Application to Globin Genes. [J]. Curr. Sci, 1994,66:309 - 314.

[5] Paul Reiners. Dynamic programming and sequence alignment [CP/OL]. 2008-05-01. http://www. ibm. com/developerworks/java/library/j-seqalign/? S_TACT=105AGX52&S_CMP=cn-a-j

Algorithm for seeking all LCS by matrix search

GONG Jie-qing

(Colle. of Soft. Engn., Southeast University, Nanjing 211189, China)

Abstract: After obtaining the two-dimensional array by dynamic programming algorithm, all ramifications using the method of matrix search can be found by this algorithm for seeking all LCS. This algorithm decreases the Omega, which is considered as index magnitude, to max{O(cmn), O(ck)}. And then, the validity and efficiency are proved.

Key words: LCS; matrix search; algorithm