项目一：MSA

人工智能　CS410　2021年秋季

姓名：李子龙　学号：518070910095　日期：2021 年 10 月 15 日

目录

1 题目

1.1 Topic

Implement three algorithms to solve multiple sequence alignment (MSA) problems.

1.2 Requirements

(1) Implement dynamic programming (DP) algorithm to find the optimal solution.
(2) Implement A-star (A*) algorithm to find the optimal solution.
(3) Implement genetic algorithm to find the optimal/suboptimal solution.

1.3 Rules

表 1: Cost Matrix

	Match $\alpha(p,p)$	Mismatch $\alpha(p,q)$	Gap δ
Cost	0	3	2

The table above shows the pairwise cost matrix. For multiple sequence alignment, the cost should be calculated in a cycle pairwise manner. Note that GAP-GAP is a match and should be considered as 0 cost. For every query, find the best alignment(s) in the database with the lowest cost.

2 动态规划算法

2.1 双序列比对

在算法与复杂性课程[1]里，已经提到了双序列比对的动态规划算法，如图 1 所示，双序列比对对于一个状态只需要考虑三个临近状态的转移，分别是对齐α，间隔 δ_x、δ_y，转换行动如表 2 所示。对于每一个状态，都需要考虑经过哪一条路径消耗最小，于是就有了如算法 1 的动态规划状态转移方程。

Algorithm 1: 双序列比对动态规划 MSA

Input: $x_1x_2\cdots x_m, y_1y_2\cdots y_n, \alpha, \delta$
Output: minimum cost

for $i \leftarrow 0$ *to* m **do** $M[i,0] = i\delta$;
for $j \leftarrow 0$ *to* n **do** $M[0,j] = j\delta$;
for $i \leftarrow 1$ *to* m **do**
  for $j \leftarrow 1$ *to* n **do**
    $M[i,j] = \min(\alpha[x_i,y_j] + M[i-1,j-1], \delta + M[i-1,j], \delta + M[i,j-1])$;
return $M[m,n]$;

表 2: 双序列行动坐标变换表

	i	j
α	$+1$	$+1$
δ_x	0	$+1$
δ_y	$+1$	0

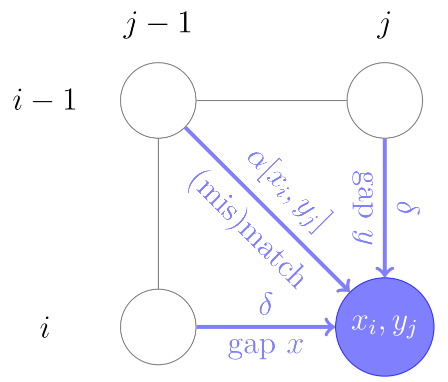

图 1: 动态规划双序列比对

2.2 多序列比对

对于三序列比对，情况就复杂地多，需要同时考虑七条路径。

表 3: 三序列行动坐标变换表

	k	j	i
$\alpha_x\delta_y\delta_z$	0	0	1
$\delta_x\alpha_y\delta_z$	0	1	0
$\delta_x\alpha_y\alpha_z$	0	1	1
$\delta_x\delta_y\alpha_z$	1	0	0
$\alpha_x\delta_y\alpha_z$	1	0	1
$\alpha_x\alpha_y\delta_z$	1	1	0
$\alpha_x\alpha_y\alpha_z$	1	1	1

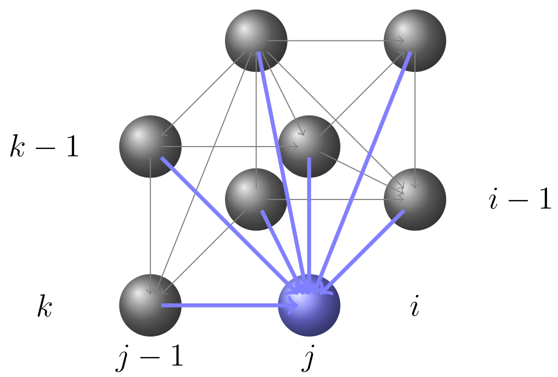

图 2: 动态规划三序列比对

可以统一化为多序列比对问题。对于 L 条序列比对，首先需要递归地初始化低维度边缘（如图 3 所示），之后余下空间其行动转换方法可以被表示为二进制从 $(\underbrace{0\cdots01}_{L\text{ digits}})_2$ 到 $(\underbrace{1\cdots11}_{L\text{ digits}})_2$ 内所有的数（最低位为第一维度），计算损耗使用上三角成对比较，规则统一为

$$\texttt{compare} = \begin{cases} 0, & (-,-)\|(p,p) \\ 2, & (p,-)\|(-,q) \\ 3, & (p,q) \end{cases}$$

并在确定每一次行动后记录路径，最后回溯路径到原点。

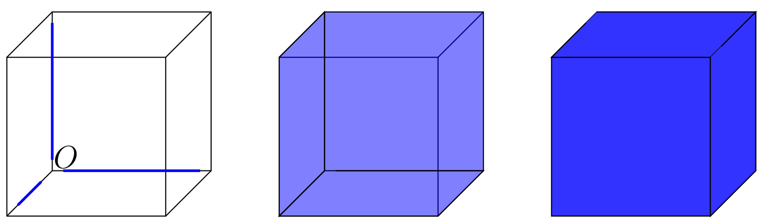

图 3: 降维递归

几乎类似于双序列比对，下面是 numpy 实现版本，虽然其速度没有使用 Python 内置的 list 版本（msa_mdp.py）的快，但是代码可读性已经与伪代码相当。

Listing 1: msa_ndp.py

```python
def editDistanceDP(S,dist:np.array=np.array([]),move:np.array=np.array([])):
    L = len(S)
    if L == 1:
        dist = np.array([i*delta for i in range(len(S[0])+1)])
        move = np.ones(shape=(len(S[0])+1), dtype=np.uint8)
        move[0] = 0  # origin is 0
        return dist, move
    if len(dist)==0:
        # initialize dist and move
        shape = tuple(len(S[l])+1 for l in range(L))
        dist = np.ones(shape=shape, dtype=np.int32)
        dist = -1 * dist        # negative means no data
        move = np.zeros(shape=shape, dtype=np.uint8)
    # calculate the lower dimension (edges)
    for s in range(L):
        slicer = tuple(0 if i==s else slice(None) for i in range(L)) # slice(None) stands for : symbol
        dist[slicer], move[slicer] = editDistanceDP(S[0:s]+S[s+1:L], dist[slicer], move[slicer]) # skip S[s]
        # configure move, insert 0 in the corresponding bit
        # Example: 4-dim xyzw xyw cube z(2) = 0, get an move 111(wyx), but with that be zero, it should be 1011.
        # REMEMBER to place the right end in the same level!
        move[slicer] = (move[slicer] >> s << (s+1)) + (move[slicer] & (2**s-1))
    # Spread the remaining space, since the edge case has been considered, the remaining space will have the same action set.
    it = np.nditer(dist, flags=['multi_index'], op_flags=["readwrite"])
    while not it.finished:
        pos = it.multi_index
        if 0 in pos:
            it.iternext()
            continue    # calculated
        ## The range of available move is 1~(2^L-1)
        minmove = np.uint8(0)
        minvalue = np.inf
        for m in range(1,2**L):
            move_vec = decodeMove(m,L)
            prev_pos = tuple(a-b for a,b in zip(pos,move_vec))
            penalty = comparelist([S[a][p] if move_vec[a]==1 else "-" for a,p in enumerate(prev_pos)])
            moved_dist = dist[prev_pos] + penalty
            if moved_dist < minvalue:
                minmove = m
                minvalue = moved_dist
        it[0] = minvalue
        move[pos] = minmove
        it.iternext()
    return dist, move
```

2.3 运行时间

如果字符串平均长度为 l，该算法 L 维字符串的复杂度为：

$$O_S = \prod_{i=1}^{L} \texttt{len}(S[i]) = O(l^L)$$

对于该问题，有 m 个待比对序列，n 个数据库项目，总时间复杂度为：

$$mC_n^{L-1}O_S \approx mC_n^{L-1}l^L$$

实际运行时间如表 4，在服务器上运行时间如下。

表 4: 动态规划运行时间

	双序列		三序列	
	朴 素 实 现 msa_dp.py	list实 现 msa_mdp.py	list 实 现 msa_mdp.py	numpy 实 现 msa_ndp.py
运行时间	29s	1min	24h	∼36h

3 A* 算法

3.1 算法描述

A* 算法会从后继结点中首先扩展评估函数 $f(n) = g(n) + h(n)$ 最小的结点，如果 $h(n)$ 的选择满足可满足启发式和一致性的性质，就可以找到按照贪婪算法的思想找到最优解。

这里将会非常乐观地估计剩下的字符串剩余部分都可以完美匹配，只会剩余间隔损耗。对于状态为 n 的启发函数就可以被定义为轮换剩余长度差的和

$$\delta \sum_{cyc} |(l_1 - \mathrm{pos}[i]) - (l_2 - \mathrm{pos}[j])| \geq \delta \left(L \max a_i - \sum_i a_i \right) = h(n)$$

其中

$$a_i = l_i - \mathrm{pos}[i]$$

不等式容易从下面图 4 的可视分析中论证，这样选择的 $h(n)$ 满足**可满足启发式**。

```
AAA00000000    AAA00000000
AAAAAA00000    AAAAAA00000
AAAAAAAAAAA    AAAAAAAAAAA
```

图 4: 每一个间隔都至少贡献了一次

之后来证明**一致性**。对于 A* 算法而言，其下一步的定义如图 6 和 7 所示。此处每一步的损耗都会大于等于0，而这种最好情况只会在全部序列都减少了 1 长度才会产生（超体对角线），这种情况下$h(n) = h(n')$；由于坐标至少在某一维度上增加了 1，一旦产生了间隙，就会有至少 2 的损耗，但是启发函数只会对应地减少 1，所以这个函数将满足一致性：

$$h(n) \leq c(n, a, n') + h(n')$$

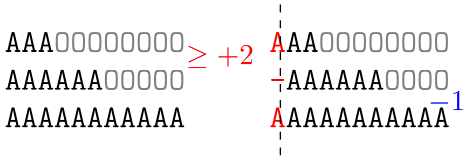

图 5: 前进一步的不等式贡献

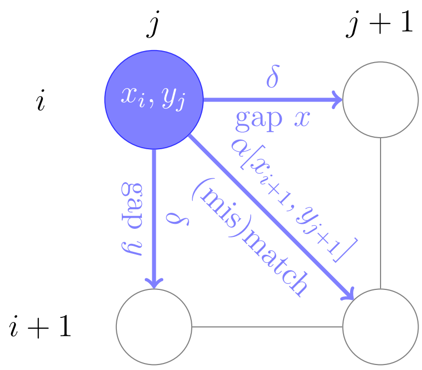

图 6: A* 双序列比对

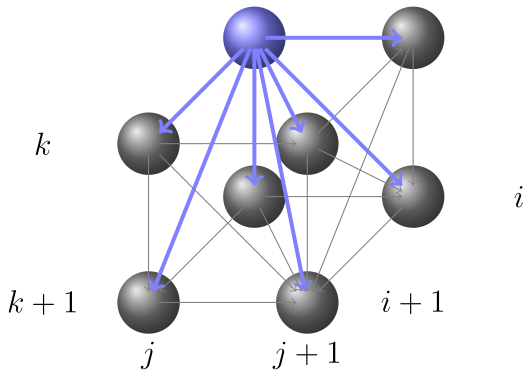

图 7: A* 三序列比对

伪代码描述如算法 2 所示，其中可选行动随着坐标的不同可能会被限制，这样就会首先扩展评估函数最小的结点。

Algorithm 2: A* 多序列比对

Input: L个字符串列表S,α,δ
Output: minimum cost

```
pos←origin;
cost ←0;
repeat
    minmove ←NULL;
    mineval ← ∞;
    foreach avaliable move do
        n ←pos+avaliable move;
        f(n) = cost + comparelist(avaliable move) + h(n);
        if f(n) < mineval then
            minmove ←avaliable move;
            mineval ← f(n);
    pos←pos+minmove;
    cost ← cost + comparelist(minmove);
until pos=finish;
return cost;
```

3.2 运行时间

该算法的时间复杂度，对于 L 个字符串（平均长度为 l），最多进行 $\sum_i l_i$ 步，每一步的分支因子为 2^L-1，单个实例需要花费时间

$$(2^L-1)\sum_i l_i = L(2^L-1)l = O(l)$$

参考文献

[1] XIAOFENG G. Algorithm & complexity class lab 06[EB/OL]. 2021. https://github.com/LogCreative/AlgAndComplexity/blob/master/Lab06/Code-SequenceAlignment.cpp.