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 21 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

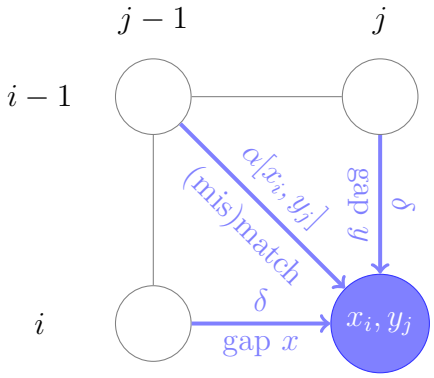

图 1: 动态规划双序列比对

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

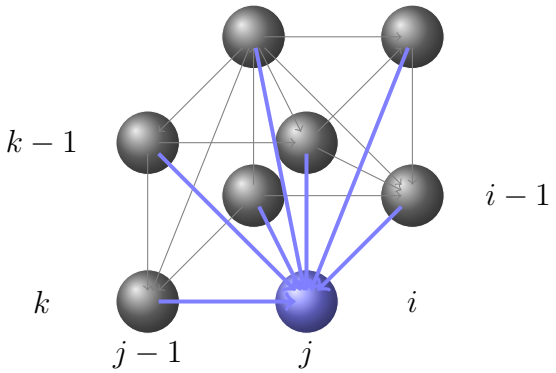

图 2: 动态规划三序列比对

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，注意附加高维度的间隙），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。

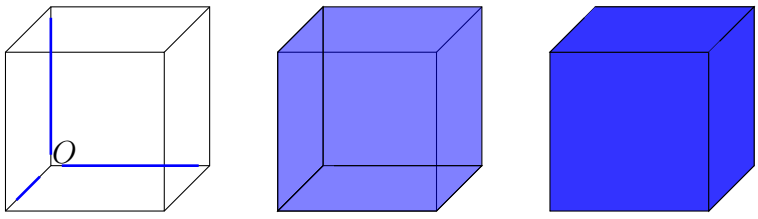

图 3: 降维递归

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

```
11 def editDistanceNDP(S,dist:np.array=np.array([]),move:np.array=np.array([])):
12     L = len(S)
13     if L == 0:
14         return np.array([0]), np.array([0])
15     global fdim
16     if len(dist)==0:
17         # initialize dist and move
18         shape = tuple(len(S[l])+1 for l in range(L))
19         dist = np.ones(shape=shape, dtype=np.int32)
20         dist = -1 * dist        # negative means no data
21         move = np.zeros(shape=shape, dtype=np.uint8)
22         fdim = L
23     # calculate the lower dimension (edges)
24     for s in range(L):
25         slicer = tuple(0 if i==s else slice(None) for i in range(L)) # slice(
       None) stands for : symbol
26         dist[slicer], move[slicer] = editDistanceNDP(S[0:s]+S[s+1:L], dist[
       slicer], move[slicer]) # skip S[s]
27         # configure move, insert 0 in the corresponding bit
28         # Example: 4-dim xyzw xyw cube z(2) = 0, get an move 111(wyx), but with
       that be zero, it should be 1011.
29         # REMEMBER to place the right end in the same level!
30         move[slicer] = (move[slicer] >> s << (s+1)) + (move[slicer] & (2**s-1))
31     # Spread the remaining space, since the edge case has been considered, the
       remaining space will have the same action set.
32     it = np.nditer(dist, flags=['multi_index'], op_flags=["readwrite"])
33     while not it.finished:
34         pos = it.multi_index
35         if 0 in pos:
36             it.iternext()
37             continue    # calculated
38         ## The range of available move is 1~(2^L-1)
39         minmove = np.uint8(0)
40         minvalue = np.inf
41         for m in range(1,2**L):
42             move_vec = decodeMove(m,L)
43             prev_pos = tuple(a-b for a,b in zip(pos,move_vec))
44             penalty = comparelist([S[a][p] if move_vec[a]==1 else "-" for a,p in
        enumerate(prev_pos)]+["-" for i in range(fdim - L)])  # the term is
       required since the higher dim will be gapped.
45             moved_dist = dist[prev_pos] + penalty
46             if moved_dist < minvalue:
47                 minmove = m
48                 minvalue = moved_dist
49         it[0] = minvalue
50         move[pos] = minmove
51         it.iternext()
52     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	48h	~72h

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之下界

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

其中

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

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**。

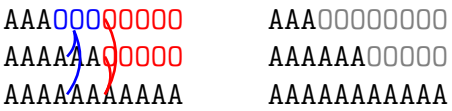

图 4: 每一个间隔都至少贡献了一次

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：

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

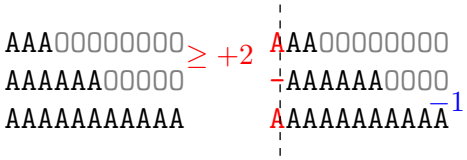

图 5: 前进一步的不等式贡献

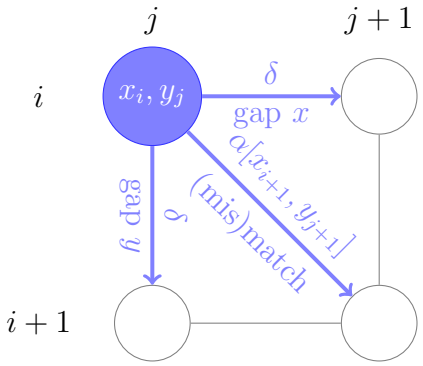

图 6: A* 双序列比对

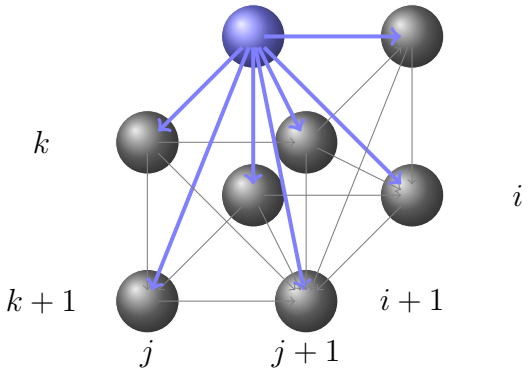

图 7: A* 三序列比对

伪代码描述如算法 2 所示[2]，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。

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），每一步的分支因子为 2^L-1，单个实例需要花费时间

$$< l^L \times (2^L - 1) = O(l^L)$$

因为这是一个树状结构的图，所以一定能够找到路径。扫描的节点数（第一项）要比动态规划小（如图 8 所示，由 `compare.ipynb` 生成），在常数级上会因为分支因子的多少而产生一定的差距。最多不会超过$\Theta((2^L-1)^{Ll})$（实际上应当多项式时间内即可，这个是 A* 的最差复杂度）。

表 5: A* 运行时间

	双序列比对	三序列比对
运行时间	2min	100h

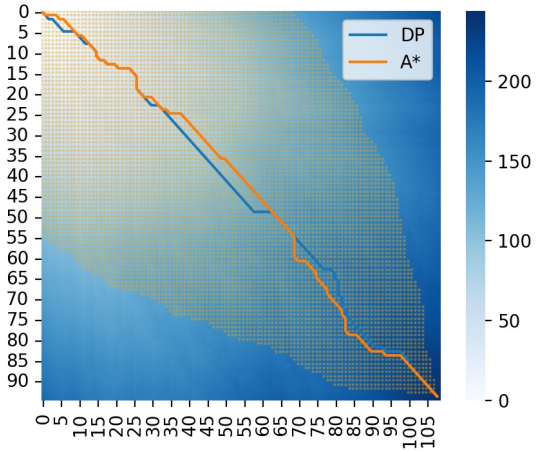

图 8: 动态规划与A*扫描节点数上的比较

Algorithm 2: A* 多序列比对

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

```
dist[·] ← ∞;
move[·] ← 0;
dist[start] ← 0;
move[start] ← 0;
openSet ← Min-Heap();
openSet[start] = h(start);
closeSet ← {};
repeat
    current ← openSet.pop();
    if current=finish then
        return dist[current];
    closeSet.add(current);
    foreach available move of current do
        n ←pos+available move;
        g(n) = dist(n) + comparelist(available move);
        if g(n) < dist[n] then
            move[n] ←available move;
            dist[n] ← g(n);
            if n not in closeSet then
                openSet[n] ← g(n) + h(n);
until openSet is empty;
return ∞;
```

3.3 改进

但是从图 8 我们可以看到这个版本的 A* 仅筛除了 30% 左右的节点，导致其运行速度依然很慢。需要选择一个更好的启发函数 $h(n)$ 以解决这个问题。

4 遗传算法

4.1 算法描述

初始化种群 在构造随机状态的时候，首先需要考虑构造出的比对序列长度。经验[3]告诉我们：

$$l = k \max(l_1, l_2, \cdots, l_n)$$

中的缩放因子k取$(1.2, 1.5)$区间内较为合适。然后分别向字符串插入对应数量的空格以对齐。不需要考虑同一位置全为空隙的情况，因为这样这个位置的损耗为 0。

适应度函数 将适应度函数定义为

$$fitness(n) = l\alpha\frac{L(L-1)}{2} - cost(n)$$

表达与完全不匹配的距离。这个值越大，表明损耗越小，越有适应性。

杂交算子 均一化适应度函数后，按照对应概率选择亲本进行繁殖。繁殖的后代每一个字符串将会随机地选择其中一个亲本的对应性状。

突变算子 选中的子代随机选择一个字符串，移动其中一个间隙的位置。

具体描述如算法 3 所示。种群数目被设定为 1000，突变率被设置为 1 %。

Algorithm 3: 遗传算法多序列比对

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

```
population ←initPopulation(S);
repeat
    new_population ← ∅;
    calculate pop_fitness for population;
    if max pop_fitness >threshold then break;
    for k ← 1 to pop_size do
        p1, p2 ← choices from population based on the pop_fitness;
        child ←crossover(p1, p2);
        if mutation is triggered then child ←mutation(child);
        new_population.append(child);
    population ← new_population;
until max time is out;
return the best individual in population;
```

4.2 运行时间

运行时间主要取决于设定的时间阈值。对于双序列比对被设定为 5s，对于多序列比对被设定为 90s，并不一定得到最优解，因为繁衍可能不够完善，有可能得到次优解。

表 6: 遗传算法运行时间

	双序列比对	三序列比对
运行时间	45min	∼

5 运行结果与结论

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.

[2] Wikipedia contributors. A* search algorithm — Wikipedia, the free encyclopedia[EB/OL]. 2021. https://en.wikipedia.org/w/index.php?title=A*_search_algorithm&oldid=1040995101.

[3] GONDRO C, KINGHORN B P. A simple genetic algorithm for multiple sequence alignment [J]. Genetics and molecular research, 2007, 6(4): 964-982.