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 22 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

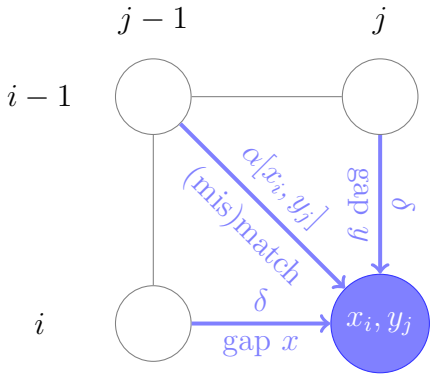

图 1: 动态规划双序列比对

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

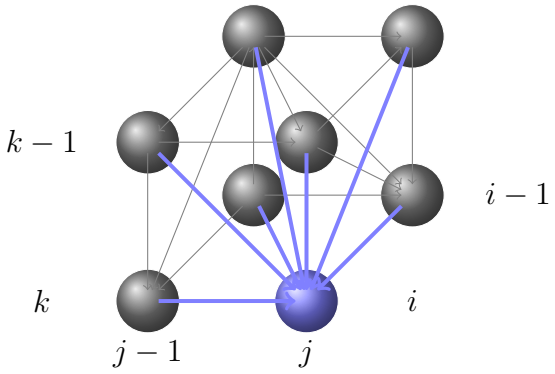

图 2: 动态规划三序列比对

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，高维度以间隙填充表示），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。

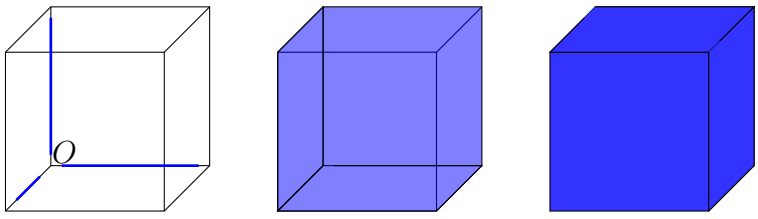

图 3: 降维递归

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

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

$$O_S = (2^L - 1)\prod_{i=1}^{L} \texttt{len}(S[i]) = O(l^L) \tag{1}$$

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	3s	33s	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**。

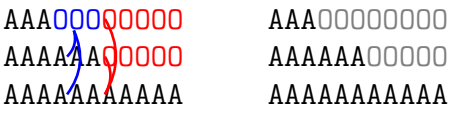

图 4: 每一个间隔都至少贡献了一次

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：

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

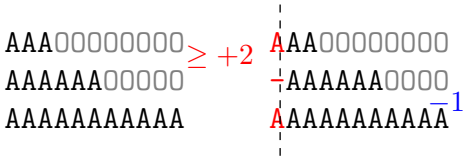

图 5: 前进一步的不等式贡献

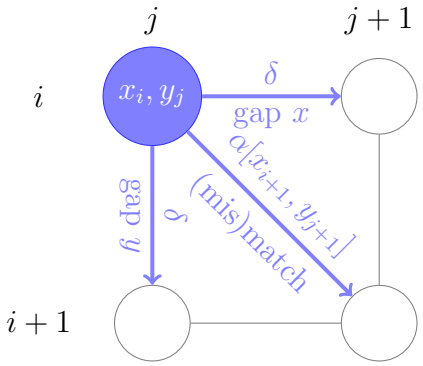

图 6: A* 双序列比对

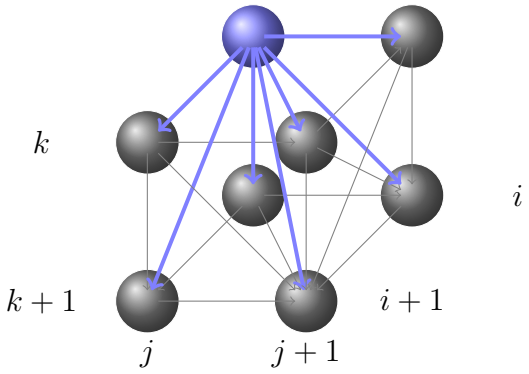

图 7: A* 三序列比对

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

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

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

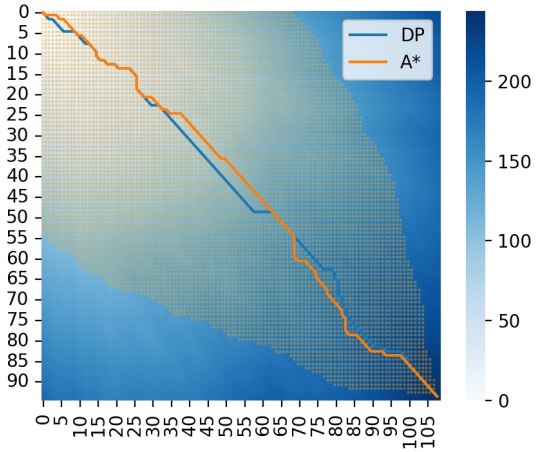

图 8: 动态规划与A*扫描节点数上的比较

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。

该部分算法实现于 msa_hastar.py 。启发函数可以使用轮换低维度反向损耗和以计算下界[3]。根据表 4，二维动态规划是非常快的，尤其是使用专用加速的实现方法 msa_dp.py ，那么启发函数就可以定义为二维反向损耗轮换和

$$h(n) = \sum_{i=1}^{L} \sum_{j=i+1}^{L} R_{i,j}[\mathtt{pos}[i], \mathtt{pos}[j]]$$

式中 $R_{i,j}$ 为字符串反置$\overline{S_i}$和$\overline{S_j}$的距离矩阵$M_{\bar{i},\bar{j}}$（由二维动态规划算法 1 定义）的对角反置，矩阵中每个点表示该位置距离匹配完成（右下角）的最小损耗，即

$$R_{i,j} = [M^T_{\bar{i},\bar{j}}[0], M^T_{\bar{i},\bar{j}}[1], \cdots, M^T_{\bar{i},\bar{j}}[L]]^T$$

首先证明其**可满足性**。如图 9，如果在高维空间中剩余字符串的最短路径在低维度上的投影就是对应低维度上的低维最短路径，那么这个启发函数就是剩余距离（轮换定义的不同计算方法）。但大部分情况下，这种投影并不是对应的，在低维度上的投影很有可能会绕一些远路而比纯粹的低维度最短路径要长。所以轮换低维度最小损耗和是真正剩余损耗的下界。

$$h(n) \leq \mathtt{cost} - g(n)$$

再证明其**一致性**。如图 10，采用**反证法**，如果不满足一致性，则存在一个状态，其

$$h(n) - h(n') > c(n, a, n')$$

意味着其低维最短路投影损耗之和超过了高维单步损耗。事实上，考虑不等式两边的低维度分量（因为都是轮换定义的）

$$\sum_k M_k(n) - \sum_k M_k(n') > \sum_k c_k(n, a, n')$$

即

$$\sum_k (M_k(n) - M_k(n') - c_k(n, a, n')) > 0$$

下面再用**反证法**证明$0 \geq M_k(n) - M_k(n') - c_k(n, a, n')$。由于在计算低维度$M$矩阵时，如果$c_k(n, a, n') < M_k(n) - M_k(n')$，对$M$来说，$n$是$n'$的后继节点（反置所致），由于 $c_k(n, a, n') + M_k(n') < M_k(n)$，那么经过$n'$是$n$的最短路径，应当有$c_k(n, a, n') + M_k(n') = M_k(n)$产生了矛盾。所以

$$0 \geq \sum_k (M_k(n) - M_k(n') - c_k(n, a, n'))$$

至此，两个式子产生了矛盾，所以$h(n)$符合一致性。

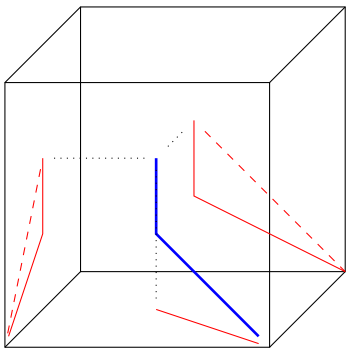

图 9: 改进后启发函数可满足性

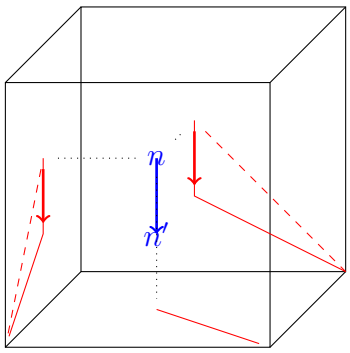

图 10: 改进后启发函数一致性

大致算法如算法 3 所示。在时间复杂度上的分析，需要首先计算上述的矩阵，之后见图 11 可以看到基本上筛除了 90% 以上的节点，对于真正的 A* 部分而言几乎就是线性时间。

$$\underbrace{\sum_{i=1}^{L}\sum_{j=i+1}^{L} 3l_i l_j}_{DP} + \underbrace{(2^L - 1)\sum_{i=1}^{L} l_i}_{A^*} \approx 3C_L^2 l^2 + (2^L - 1)Ll = O(l^2) \tag{3}$$

Algorithm 3: 改进后 A* 多序列比对

1 根据算法 1，计算每对反转字符串的距离矩阵M，并对角线反转得到 R;
2 设定启发函数$h(n)$为对应坐标轮换对应的R矩阵值的和;
3 按照算法 2，进行 A* 运算;

表 6: 改进后 A* 运行时间

	双序列比对	三序列比对
运行时间	7s	3h30min

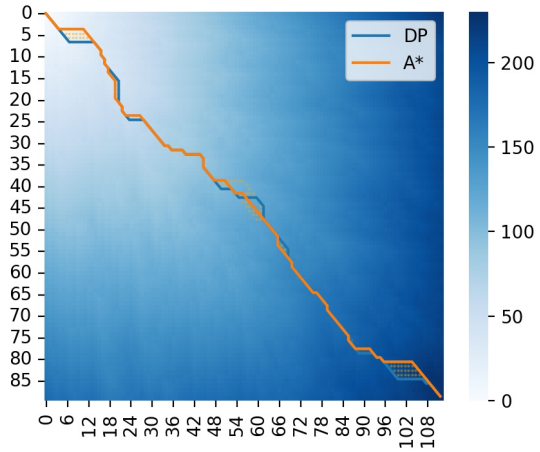

图 11: 动态规划与改进后A*扫描节点数比较

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[4]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 突变算子只能改变间隙，不能够修改字符（否则字符串比对就会不完整）。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 4 所示。种群数目被设定为 1000，突变率被设置为 1 %。

Algorithm 4: 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 7: 遗传算法运行时间

	双序列比对	三序列比对
运行时间	45min	∼

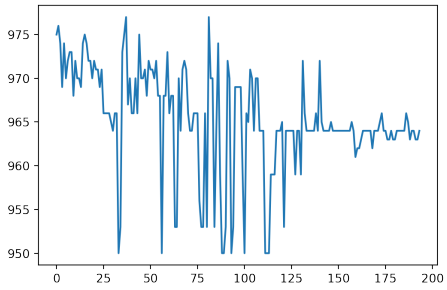

图 12: 遗传算法收敛情况

4.3 改进

如图 12 所示，现有的基本遗传算法收敛速度是相当慢的。为了让遗传算法的收敛速度更快，需要对初始化种群的方法进行改进。

采用二维动态规划出的匹配序列进行初始化种群[4]，现在的字符串长度

$$l = k \max(l_1, l_2, \cdots, l_n) + x \tag{4}$$

式子中 x 为初始偏移最大值，取 $1.2 \max_i l_i$，算法 5 给出了初始化方法。

Algorithm 5: 基于动态规划的种群初始化

1 对每一对字符串根据算法 1 计算出最佳比对;
2 对于每一种群，每一个个体选择其中对应的一种匹配串，增加随机数量（不超过 x）的前缀间隔，后缀补齐间隔为预定字符串最大长度 l;

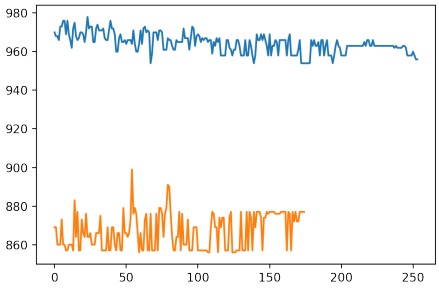

图 13: 改进初始化种群后

这种方法可以进行预对齐，图 13 可见其起点变得更低了。但是其减少速度却没有明显变化。这个时候需要修改适应度函数，基于内卷的方法，将损耗最大的设置为最差的，并被**筛除**，当前轮其余个体在损耗上与之的差值作为适应度函数。这样就是 1:1 对应的局部竞争，选择优秀个体繁衍后代，每一轮的标准都会有所不同。并且提高突变概率为 10%，以提高多样性。图 14 显示了这种方法的更快的收敛速度。

$$fitness(i) = \max_i(\texttt{cost}_i) - \texttt{cost}_i$$

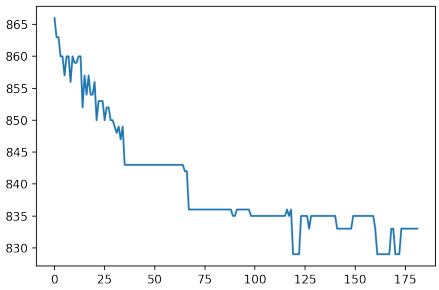

图 14: 改进适应度函数后

5 运行

5.1 运行框架

为了更好地进行代码复用，本工程采用如表 8 的框架。

表 8: 运行框架

crosstest.py (测试) main.py (计算)					
msa_dp.py	msa_mdp.py	msa_ndp.py	msa_astar.py	msa_hastar.py	msa_ga.py
msa_util.py					

运行测试是为了使用一个样例进行交叉测试，以避免大型计算后才发现一些算法错误。直接运行主文件，会被询问用什么算法完成，并选择计算几维的样例。

5.2 运行结果

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] HATEM M, RUML W. External memory best-first search for multiple sequence alignment [J/OL]. Proceedings of the AAAI Conference on Artificial Intelligence, 2013, 27(1): 409-416. https://ojs.aaai.org/index.php/AAAI/article/view/8626.

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