人工智能 Project1

目录

1 算法简介

1.1 动态规划（DP）

对于双序列匹配问题，每一步我们都有三种选择：序列A插入空格、序列B插入空格、序列AB均不插入空格，但我们做出选择之后，剩下的序列成为了一个全新的双序列匹配问题。

动态规划算法使用表格记录下所有子问题的答案，用于在碰到相同子问题时快速给出答案，以此来降低搜索的时间复杂度。

在双序列匹配算法的实现中，我们使用 `numpy` 模块创建一块二维表格，表格的长宽分别等于两条序列的长度。为了方便后续A*算法启发函数的实现，我们的DP算法从序列末尾开始匹配，不同的匹配方向可能会影响序列中GAP的位置，但不会影响最终的匹配代价。

算法运行时，我们依次填充整个表格，在本例中，我们会从表格右下角开始、以斜角顺序进行填充，对于(x,y)处的单元格，其状态状态转移方程如下：

$$c(x,y)=\begin{cases}c(x+1,y)+GAP\_COST\\c(x,y+1)+GAP\_COST\\c(x+1,y+1)+MISMATCH\_COST, & \text{if } S_1[x]\neq S_2[y]\\c(x+1,y+1), & \text{if } S_1[x]=S_2[y]\end{cases}$$

每个单元格的填充会在所有的可能代价之间选择最低的值，最终填充完整个表格之后，$(0,0)$单元格的值即为最优匹配代价。

斜角填充顺序可以保证每个问题被求解时，所有与其相关的子问题都已经完成计算，因此求解单个问题的均摊时间复杂度为$O(1)$。记单序列的长度为n，双序列匹配动态规划算法的时间复杂度与空间复杂度均为$O(n^2)$。

对于多序列匹配问题，也可以使用相似的方法，表格的维数需与序列数一致，因此，三序列匹配的时间复杂度与空间复杂度为$O(n^3)$。可以看出，动态规划算法的复杂度随序列数增加呈指数关系增加。

1.2 A*算法

A*算法是一种启发式搜索算法，相较无信息搜索遍历整个解空间，启发式搜索算法利用当前节点状态与目标节点状态的差异对当前节点到目标节点的距离进行估计，由此来降低搜索量。

具体到A*算法，我们需要定义一个启发函数$h(n)$，该函数可以估计当前节点到目标节点的距离；令$f(n)=g(n)+h(n)$，$g(n)$代表从初始节点到达当前节点的开销，优先搜索$f(n)$值较小的节点，有助于更快到达目标节点。对于启发函数$h(n)$，只要$h(n)\leq h^*(n)$，便称之为可信的，使用可信启发函数的A*算法一定能得到最优解。

在双序列匹配问题中，每个节点用一个二元组(x,y)代表当前节点对应的序列位置，每个非边界节点可以展开为三个节点：$(x+1,y),(x,y+1),(x+1,y+1)$。对于每个节点，在计算出预估代价$f(n)$之后，将其放入优先队列，每次从队列中取出预估代价最小的节点进行展开，直到取出目标节点。双序列匹配问题中，我们的启发函数$h(n)$定义为两条序列剩下部分长度只差乘上匹配空隙的代价，这是一个相当粗略的估计，但是对于双序列匹配问题已经足够了。

对于我们实现的双序列匹配A*算法，其启发函数复杂度为$O(1)$。在最差的情况下，解空间中所有的节点都被刚好展开了一次（使用一个布尔数组来保证节点不会被重复展开），假设序列长度为$O(n)$，总共将有$O(n^2)$个节点被展开，优先队列入队$O(3n^2)$次。在最好的情况下，该算法可在$O(n)$个节点后到达目标节点。因此，该算法在最差情况下的时间复杂度为$O(n^2\log n)$，最好情况下复杂度为$O(n\log n)$。

对于三序列匹配，我们将采用一种更为准确的启发函数，首先引入一个定理：

> 对于序列s_1,s_2,s_3，它们的三序列匹配代价大于等于两两匹配代价之和，即$Cost_3(s_1,s_2,s_3)\geq\sum_{1\leq i<j\leq 3}Cost_2(s_i,s_j)$

于是，我们将在三序列匹配开始前，对三条序列两两使用动态规划进行双序列匹配，若我们从末尾开始动态规划匹配，表格中(x,y)处的值正好对应了两条分别从x与y处开始的子序列的匹配代价。因此，三序列匹配过程中，启发函数时间复杂度之和为$O(3n^2)$。

与双序列匹配的推导类似，我们可以推出基于A*算法的三序列匹配在最差情况下的时间复杂度为$O(n^3 \log n)$。最好情况下，节点展开复杂度总和为$O(n \log n)$，考虑启发函数，整体的时间复杂度为$O(n^2)$。

1.3 遗传算法（GA）

遗传算法如其名字所述，是一种灵感源自自然界中生物遗传的算法。在自然界中，生物在遗传过程中，遗传物质会发生变异、交叉互换等变化，每一个个体会表达出不同的性状，这些性状对自然环境有不同适应性，适应环境的个体存活下来继续产生后代，引发整个群体的进化。

遗传算法正是要模拟这种进化过程。我们每一个个体有一段编码序列，通过对编码序列进行遗传算子操作，可以组合得到新的个体。我们也有一个适应度函数用于为每个个体打分，得分较高的个体有更高的概率被保留。

在多序列匹配中，我们将每一条序列中空位（即GAP）的位置作为遗传编码序列，并且定义了三种遗传算子：

- 交叉互换：两条编码序列从某一个位置切开并重新组合成为新序列
- 变异：编码序列中一位数字变为一个随机值
- 位移：编码序列中一位数字变为一个随机值，且新旧值差距在限定范围内

为分析遗传算法的时间复杂度，我们记种群规模为S，进化代数为G，用于匹配的序列长度为n，匹配序列数为m。其中，S与G为用户手动设置的超参数，与输入问题规模无关。

- 遗传算子操作：遗传算子只对编码序列进行操作，考虑到编码序列往往远小于匹配序列长度，对于单个个体的遗传算子操作，其复杂度可以视为$O(1)$。
- 适应度函数：对于每一个个体，适应度函数需要首先将空位嵌入序列（序列展开），然后再成对计算匹配代价。序列展开的复杂度为$O(mn)$，就算匹配代价的复杂度为$O(m^2n)$。
- 选择：在此过程中，我们将**可重复地**从S个个体中加权选出S个个体，选择的时间复杂度为$O(S^2)$。

综合来看，遗传算法的时间复杂度为$O(GS(m^2n+S))$。

在实际运行中，为了取得更好的结果，可以改变结束条件，例如改为连续50代没有进化。需要指出的是，遗传算法具有随机性，每一次的运行结果与运行时间都可能有较大差异。

2 代码文件及接口

2.1 依赖

```
numpy >= 1.19.1
numba >= 0.54.1         #optional
```

2.2 文件列表

- Dataset.py：数据集封装。
- BaseSolver.py：包括求解器基类 `BaseSolver` ，测试类封装 `Tester` 。
- AStarSolver.py：A*算法双序列匹配实现。
- AStarSolver_3.py：A*算法三序列匹配实现。
- DPSolver.py：DP算法双序列匹配实现。
- DPSolver_numba.py：DP算法双序列匹配实现（使用 `Numba` 加速）。
- DPSolver_3.py：DP算法三序列匹配实现。
- DPSolver_3_numba.py： DP算法三序列匹配实现（使用 `Numba` 加速）。

- GASolver.py：遗传算法双序列匹配实现。
- GASolver_3.py：遗传算法三序列匹配实现。

所有求解器的实现代码已经内置 Tester 类调用，可直接运行对应的 .py 文件。

2.3 BaseSolver类

- 构造函数

  参数:

  - seq：参与匹配的序列。

- solve

  调用该函数后，求解器求解序列匹配问题，返回匹配代价。

- get_align_cost

  返回匹配代价。

- get_lower_evaluation

  返回匹配代价下界估计值。

- align_seq

  返回完成对齐的序列，即将空位插入到原序列中。

- clean

  清理临时数据。

2.4 Tester类

Tester 类提供了对不同算法的快速测试功能。

- 构造函数

  参数:

  - SolverClass：求解器类，应为 BaseSolver 的子类。
  - query：查询字符串。
  - single_match：bool 类型，为 True 时从数据库选择单条序列与 query 进行匹配（即双序列匹配），否则选择两条序列与之匹配（三序列匹配）。
  - allow_skip ：bool 类型，为 True 时启用下界估计优化。
  - hashbin_enabled ：bool 类型，为 True 时启用哈希表加速。
  - file：文件对象，用于保存匹配过程信息。参数留空时不保存。
  - quite： bool 类型，为 True 时开启安静模式，不输出匹配过程。

- run_test

  此函数无参数，调用后将开始测试。

3 运行结果

下列结果中，数字表示匹配代价，序列中 * 表示空位。结果按照 MSA_query.txt 中顺序排列。

3.1 动态规划

3.1.1 双序列匹配

```
112
XHAPXJAJXXXJAJ**XDJAJXXXJAPX*JAHXXXJAJ**X*DJAJ**X**XXJAJ*XXXJPPXJAJXXXHAPXJAJXXXJAJ**X**XXJAJ**X**XXJA
******KJ*XXJAJKPXKJ*J**XJKPXKJ***XXJAJKPXKJJXJKPXKJXXJAJKPXKJ*XXJAJ**KH**X*KJ*XXJAJKPXKJXXJAJKHXKJXX**
107
IPOTWJJLAB*KS*GZG*WJJKSPPOPHT**SJ*EWJHCTOOTH*RAXBKLBHWPKZULJPZKAKVKHXUIKLZJGLXBTGHOHBJLZPPSJJ*PJO
ILOTGJJLABWTSTG*GONXJ*****MUTUXSJHKWJHCT*OQHWGAGIWLZHWPKZULJTZWAKBWHXMIKLZJGLXBPAHOHVOLZWOSJJLP*O
113
I***K*BSKKKK**W*KKK*K*KKKKWWK*KKKPGK*KK*KKXXGGKRKWWKWKKPK*K*KKKXK*KKRWKMKKPK**KWPKKK***KK*PGKK***KLBKW***WKKJ
IHKKKRKKKKKKXGWGKKKPKSKKKK**KBKKKP*KHKKXKK**BSK*K*PKWKKLKSKRKKWXKPKK*BK*KKPKTSKHKKKKLADKKYPKKKOPHKKBWWLPPWKK*
148
**OPJPXJP*PMJPPMX*PP*MJ*PPXJPPOXPPXJJPJXXPXJPPOJPPMXPPOGP**PXXP*P****OM***P********PXXPPOXPPXJPQXPPBJPPPXPPX
MPPPJPX*PGP*JPPPXPPPJPJPPPXPPPPSPPJJJPPXXP*PPPPJPPPXPPXIPJMMMXPKPSVGULMHHZPAWHTHKAAHHUPAONAP*J*SWPP*J*****GA
131
ITPVKWKSKXKXUAXP*VHXVO*M*MKHYBPABLLOBGKOLLJGXZGXLSOL**AMOGKIGXBATBXMPJTCVMTAXVMPWWA***WOM**OUPHHZBITKKXLK
IPPVKBKXWXKHSA*PHVXXVOJMRAK**KPJVLLJBWKOLLJKXHGXL**LCPAJOBKPGXBATGXMPOMCVZTAXV*P**AGKXGOMJQO****LJGWGKXLQ
```

3.1.2 三序列匹配

```
290
IPZJJPL*LTHUULOSTXTJOGLKJGBLLMMPJPLUWGKOMOYJBZA*YUKOFLOSZHGBP*HXPLXKJBXKJL*AUUOJHW*TWWPQ
IPMJJ*LLLTHOULOSTMAJIGLKJPVLLXGKTPLTWWKOMOYJBZP*YUKLILOSZHGBPGWX*LZKJBSWJLPJUU*MHK*TRAP*
IPZJJ*LMLTKJULOSTKTJOGLKJOBLTXGKTPLUWWKOMOYJBGALJUKLGLOSVHWBPGWS*LUKOBSOPLOOKU***KSARPPJ
```

```
122
**IWTJBGTJGJTWGBJTPKHAXHAGJJXJJKPJTPJHJHJHJHJHJHJHJHJHJHKUTJJUWXHGHHGALKLPJTPJPGVXPLBJHH**********
WPIWTJBGTJGJTHGBJOXKHTXHAGJJXJJPPJTP**JHJHJHJHJHJHJHJHJPKUAJJUWXHGHHGALKLPJTPJPGVXPLBJHHJPK*******
**IWTJBGTJGJTWGBJTPKHAXHAGJJSJJPPJAPJHJHJHJHJHJHJHJHJHJPKSTJJUWXHGPHGALKLPJTPJPGVXPLBJHHJPKWPPDJSG
```

3.2 A*算法

3.2.1 双序列匹配

```
112
**XHAPXJAJXXX*JAJXDJAJX**XXJAPXJAHXXXJAJXDJAJX**XXJAJ*XXXJPPXJAJXXXHAPXJAJXXXJAJ**X**XXJAJ**X**XXJA
KJX***XJAJKPXKJ*JX*JKPXKJXXJA**JK*PXKJ*JX*JKPXKJXXJAJKPXKJ*XXJAJK**H**X*KJ*XXJAJKPXKJXXJAJKHXKJXX**
107
IPOTWJJLAB*KS*GZGWJ*JKSPPOPHT**SJE*WJHCTOOTH*RAXBKLBHWPKZULJPZKAKVKHXUIKLZJGLXBTGHOHBJLZPPSJJ*PJO
ILOTGJJLABWTSTG*GONXJ*****MUTUXSJHKWJHCTO*QHWGAGIWLZHWPKZULJTZWAKBWHXMIKLZJGLXBPAHOHVOLZWOSJJLP*O
113
I**K**BSKKKK**W*KKK*K*KKKKWWK*KKKPGK*KK*KKXXGGKRKWWKWKKPK*K*KKKXK*KKRWKMKKPK**KWPKKK***KK*PGKK***KLBKW***WKKJ
IHKKKRKKKKKKXGWGKKKPKSKKKK**KBKKKP*KHKKXKK**BSK*K*PKWKKLKSKRKKWXKPKK*BK*KKPKTSKHKKKKLADKKYPKKKOPHKKBWWLPPWKK*
148
**OPJPXJP*PMJPPMXPPMJP*P**XJPPOXPPXJJPJXXPXJPPOJPPMXPPOGP*P*XXP*P***O*M**PPXXP*POXPPXJP*QXPPBJP*PPXPPX
MPPPJPX*PGP*JPPPXPPPJPJPPPXPPPPSPPJJJPPXXP*PPPPJPPPXPPXIPJMMMXPKPSVGULMHHZPAWHTHKAAHHUPAONAP*JSWPPJGA*
131
ITPVKWKSKXKXUAXP*VHXVO*MM*KHYBPABLLOBGKOLLJGXZGXLSOL**AMOGKIGXBATBXMPJTCVMTAXVMPWWA**W*OM**OUPHHZBITKKXLK
IPPVKBKXWXKHSA*PHVXXVOJMRAK**KPJVLLJBWKOLLJKXHGXL**LCPAJOBKPGXBATGXMPOMCVZTAXV*P**AGKXGOMJQO**LJGW**GKXLQ
```

3.2.2 三序列匹配

```
290
IPZJJPL*LTHUULOSTXTJOGLKJGBLLMMPJPLUWGKOMOYJBZA*YUKOFLOSZHGBPH*XPLXKJBXKJL*AUUOJHW*TWWPQ
IPMJJ*LLLTHOULOSTMAJIGLKJPVLLXGKTPLTWWKOMOYJBZP*YUKLILOSZHGBPGWX*LZKJBSWJLPJUU*MHK*TRAP*
IPZJJ*LMLTKJULOSTKTJOGLKJOBLTXGKTPLUWWKOMOYJBGALJUKLGLOSVHWBPGWS*LUKOBSOPLOOKU***KSARPPJ
```

```
122
**IWTJBGTJGJTWGBJTPKHAXHAGJJXJJKPJTPJHJHJHJHJHJHJHJHJHJHKUTJJUWXHGHHGALKLPJTPJPGVXPLBJHH**********
WPIWTJBGTJGJTHGBJOXKHTXHAGJJXJJPPJTP**JHJHJHJHJHJHJHJHJPKUAJJUWXHGHHGALKLPJTPJPGVXPLBJHHJPK*******
**IWTJBGTJGJTWGBJTPKHAXHAGJJSJJPPJAPJHJHJHJHJHJHJHJHJHJPKSTJJUWXHGPHGALKLPJTPJPGVXPLBJHHJPKWPPDJSG
```

3.3 遗传算法

3.3.1 双序列匹配

```
149
XHA*PXJAJ*XXXJAJXDJAJX*XXJAPXJAHXXXJAJXDJAJXXXJAJXXXJPPXJAJXXXHAPXJAJX*XXJAJXXXJAJXXXJA
*KJ*XXJAJ*KPXKJJXJKPXKJXXJAJ*KP*XKJJXJKPXKJ*XXJAJKPXKJXXJAJKHXKJXXJAJKPXKJXXJAJKHXKJXX*
119
IPOTWJJLAB*KSGZG*WJJKSPPOPHTSJEWJHCTOOTHRAXBKLBHWPKZULJPZKAKVKHXUIKLZJGLXBTGHOHBJLZPPSJJPJO
ILOTGJJLABWTSTGGONXJ*MU*TUXSJHKWJHCTOQHWGAGIWLZHWPKZULJTZWAKBWHXMIKLZJGLXBPAHOHVOLZWOSJJLPO
159
IKBSKK*KKW*KK*KKKKKKW*W*KKKKP**GKKKKK*XX*GG****KRK*W*WKWKKPKKKKXKKKR*W*K*MKKP**KKWPKKKKKPG*KKKLBKWWKKJ
IHK*KKRKKKKKKXGWGKKKPKSKKKKKBKKKPKHKK*XKKBSKK*PKWKKLKSKRKKWXKPKKBKKKPKTSKHKKKKLADKKYPKKKOPHKKBWWLPPWKK*
180
*OP*JPX*JPPMJP*P**MXPPMJP*PXJP*POX*PPXJJPJXXPXJ*PPOJPPMXPPOGPP*XX*PPOMPP*XXPPOXPPX*JP*QXPPBJPPPXPPX********
MPPPJPX**PGPJP*P**PXPPPJPJPPPXPPPPSPPJJJPP*X*XPPPPPJPPPXPPXIPJMMM*XPKPSVGULMHHZPAW*H*THKAAHHUPAONAPJSWPPJGA
150
ITPVKWKSKXKX*UAXP*VHXVOMMKHYBPABLLOBGKOLLJGXZGXLSOLAMOGKIGXBATBXMPJTCVMTAXVMPWWAWOMOUPHHZBI*TKKXLK
IPPVKBK**XWXKHSAPHVXXVOJMRAKKPJVLLJBWKOLLJKXHGXLLCPAJOBKPGXBATGXMPOMCVZTAXVPAGKXGOMJQOLJGWG*KXLQ**
```

3.3.2 三序列匹配

```
311
IPZJJPLLTHUULOSTXTJOGLKJGBLLMMPJPLUWGKOMOYJBZA*YUKOFLOSZHGBPHXPLXKJBXKJLAU*UOJHWTWWPQ*
IPMJJLLLTHOULOSTMAJIGLKJPVLLXGKTPLTWWKOMOYJBZPYU*KLILOSZHGBPGWXLZKJBSWJLPJUUM*HKTRAP**
IPZJJLMLTKJULOSTKTJOGLKJOBLTXGKTPLUWWKOMOYJBGALJUKLGLOSVHWBPGWSLUKOBSOPLO***OKUKSARPPJ
```

```
140
**IWTJBGTJGJTWGBJTPKHAXHAGJJXJJKPJTPJHJHJHJHJHJHJHJHJHJHKUTJJUWXHGHHGALKLPJTPJPGVXPLBJHH**********
WPIWTJBGTJGJTHGBJOXKHTXHAGJJXJJ*P*PJTPJHJHJHJHJHJHJHJHJPKUAJJUWXHGHHGALKLPJTPJPGVXPLBJHHJPK*******
**IWTJBGTJGJTWGBJTPKHAXHAGJJSJJPPJAPJHJHJHJHJHJHJHJHJHJPKSTJJUWXHGPHGALKLPJTPJPGVXPLBJHHJPKWPPDJSG
```

4 性能优化

前面介绍的算法在面对双序列匹配时，尚能取得不错的性能表现，但是直接用于三序列匹配却是不现实的。对于动态规划与A*算法，其时间复杂度随着序列数增加呈指数趋势增长；同时，对于100条序列的数据集，需要完成的匹配次数也从100次增长到4950次。综合以上两点，如果继续采用遍历搜索的方法解决三序列匹配问题，消耗的时间大约相当于双序列匹配的8000倍。无疑，我们需要更多的优化手段来完成三序列匹配。

对于三种算法，我们的优化思路是大致相同的。

- 下界估计（Lower Bound Evaluate，LBE)

  与A*算法启发函数的原理类似，我们可以使用三组二维动态规划来计算匹配代价的下界，然后将下界与当前最低的匹配代价进行比较，若估计的代价下界高于当前最低的匹配代价，则可以直接跳过该组三序列匹配。

  在实际测试中，使用下界估计的方式，我们可以减少95%~98%的三序列匹配次数（详细数据见性能分析部分）。因此，下界估计是我们使用的优化方法中，效果最佳的优化方法。

- 哈希表查询（HashBin）

  如果我们在每个三序列匹配前都进行一次下界估计，总共需要进行14850次二维动态规划；然而，1条查询序列加上100条数据库序列，总共只有5050对组合。可以看出，有大量的二维动态规划计算是重复的，所以我们可以使用哈希表保存已经计算过的双序列匹配结果，以此来加速下界估计。

- 部分下界估计

  在实际匹配过程中，大多数情况下我们能从哈希表中快速得到两组双序列匹配的结果，在计算剩下的双序列匹配结果之前，可以先使用已知的双序列匹配代价计算一个部分下界，该下界相比真实匹配代价会有较大差距，但在一些情况下就已经低于了当前最低的匹配代价，因此可以更快决定是否跳过三序列匹配。

  在实际测试中，使用部分下界估计大概可以减少80%~90%的双序列匹配次数。

- 启用科学计算模块 `Numba`

  本优化方法并没有优化算法的时间复杂度，但是提高了代码的执行速度。利用科学计算模块 `Numba` ，可以将python函数编译为二进制，以期获得与C语言相仿的性能表现。该模块的优化对于动态规划这种涉及大量数组访问的算法尤其有效。

5 性能测试

由于双序列匹配复杂度不高，且我们的优化主要集中于三序列匹配，因此，我们的性能分析也集中于三序列匹配。

我们使用 `cProfile` 模块进行性能测试，保存测试结果并用 `pstats` 模块格式化输出，测试的原始结果见 `performance.md` ，结果总结如下表。

表中 `None` 表示未使用优化； `Lower Bound Evaluate(LBE)` 表示使用下界估计优化； `HashBin` 表示使用哈希表查询与部分下界估计； `Numba` 表示使用numba模块对求解函数进行加速； `Numba(Cached)` 表示使用numba模块且启用numba的编译缓存。

表中 `3MSA queries` 、 `2MSA queries` 、 `Hash queries` 分别表示3序列匹配、2序列匹配、哈希查询，括号外的数字表示调用次数，括号内表示运行时间；如果使用了numba，则会标记两个时间，第一个时间代表python包裹函数执行时间，第二个时间为numba内执行时间，两者时间差主要由numba编译、解释器上下文切换、数据结构初始化造成。

5.1 DP算法三序列匹配性能测试

- 序列一

Optimization	3MSA queries	2MSA queries	Hash queries	Elapsed Time
None(Baseline)	4950	-	-	~7h
Lower Bound Evaluate	18 (164.3s)	14850(39.5s/38.7s)	-	164.3s
LBE+HashBin	18 (123.8s)	797 (1.98s/1.73s)	14850 (23ms)	125.8s
LBE+HashBin+Numba	18 (15.3s/12.2s)	797 (2.00s/1.75s)	14850 (23ms)	17.3s
LBE+HashBin+Numba(Cached)	18 (11.9s/11.9s)	797 (1.82s/1.58s)	14850 (21ms)	13.7s

- 序列二

Optimization	3MSA queries	2MSA queries	Hash queries	Elapsed Time
None(Baseline)	4950	-	-	~8h
Lower Bound Evaluate	8 (56.6s)	14850(42.0s/41.8s)	-	99s
LBE+HashBin	8 (54.9s)	304 (913ms/667ms)	14850 (21ms)	55.8s
LBE+HashBin+Numba	8 (9.2s/6.2s)	304 (909ms/673ms)	14850 (22ms)	10.1s
LBE+HashBin+Numba(Cached)	8 (6.6s/6.6s)	304 (945ms/703ms)	14850 (22ms)	7.6s

5.2 A*算法三序列匹配性能测试

- 序列一

Optimization	3MSA queries	2MSA queries	Hash queries	Elapsed Time
None(Baseline)	4950	-	-	~70min
Lower Bound Evaluate	18 (3.23s)	14850(40.7s/39.7s)	-	44.2s
LBE+HashBin	18 (2.78s)	822 (1.91s/1.76s)	14850 (22ms)	4.70s

- 序列二

Optimization	3MSA queries	2MSA queries	Hash queries	Elapsed Time
None(Baseline)	4950	-	-	~70min
Lower Bound Evaluate	8 (1.02s)	14850(42.5/41.7s)	-	42.7s
LBE+HashBin	8 (1.00s)	317 (0.93s/0.70s)	14850 (20ms)	1.95s

5.3 遗传算法三序列匹配性能测试

由于遗传算法结果具有随机性，且我们的LBE优化表现依赖于先前匹配代价，因此遗传算法的耗时也具有随机性，测试结果仅供参考。

- 序列一

Optimization	3MSA queries	2MSA queries	Hash queries	Elapsed Time
None(Baseline)	4950	-	-	~50min
LBE+HashBin	104 (63.8s)	1151 (2.91s/2.58s)	14850 (50ms)	68.4s

- 序列二

Optimization	3MSA queries	2MSA queries	Hash queries	Elapsed Time
None(Baseline)	4950	-	-	~50min
LBE+HashBin	48 (34.1s)	533(1.50s/1.22s)	14850 (48ms)	37.0s

6 总结

动态规划是最经典的路径搜索算法之一，尽管搜索量较大，但是数据结构简单、数据间的依赖性关系明确，可以运行在基于SIMD的并行运算架构中。A*算法是游戏引擎中最常用的路径搜索算法，一个优秀的启发函数可以极大程度地提高A*算法性能，甚至将其搜索节点数降低至$O(n)$级别。遗传算法思路起源于自然界，无法保证结果最优，但是可以比其他算法更好地应对巨大输入规模的问题，鉴于其个体易于编码且个体之间完全没有相关性，可以运行在大规模分布式计算节点中。

在本项目中，我基于动态规划、A*算法、遗传算法解决了双序列匹配与三序列匹配问题。鉴于不经优化的三序列匹配时间开销过大，我对三序列匹配算法进行了一系列优化，使得三序列匹配速度提高了数十至数千倍，最终得到了运行时间可接受的解决方案。

有意思的是，尽管我的优化方案取得相当好的效果，但这些优化方式并没有改变原有算法的复杂度上限，算法优化前后的大O复杂度并没有变化。这次经历告诉我，在实际情况下，哪怕是对常数的优化也是相当有意义的。