算法设计与分析实验报告

- 姓名：陈颖妍
- 学号：23300240033

- 实验序号：lab2
- GitHub网站地址：https://github.com/bala-2022/DNA
- GitHub分支：master

算法伪代码

- 主算法伪代码

```
FUNCTION main():
    ref ← 从"reference.txt"读取参考序列
    query ← 从"query.txt"读取查询序列
    alignment ← alignSequences(query, ref)
    输出 formatOutput(alignment)

FUNCTION alignSequences(query, ref):
    // 第一阶段：主锚点查找
    anchors ← findAnchors(query, ref)
    extendAnchors(anchors, query, ref)
    primaryChain ← findOptimalChain(anchors)

    // 第二阶段：二级锚点查找
    secondaryAnchors ← findSecondaryAnchors(primaryChain, query, ref)
    IF secondaryAnchors不为空 THEN
        extendAnchors(secondaryAnchors, query, ref)
        primaryChain ← primaryChain + secondaryAnchors
        对primaryChain按q_start排序
        primaryChain ← findOptimalChain(primaryChain)

    RETURN processGaps(primaryChain, query, ref)
```

- 关键函数伪代码

```
FUNCTION findAnchors(query, ref):
    K ← selectKmerSize(ref.length)

    // 并行构建k-mer索引
    kmerMap ← 空哈希表
    并行处理：
        FOR 每个线程分配的ref块 DO
            FOR i从start到end-K+1 DO
                kmer ← ref.substr(i, K)
                将i添加到kmerMap[kmer]
```

```
    anchors ← 空列表

    // 查找正向匹配
    FOR i从0到query.length-K DO
        kmer ← query.substr(i, K)
        IF kmer在kmerMap中 THEN
            FOR 每个r_pos IN kmerMap[kmer] DO
                添加Anchor(i, i+K, r_pos, r_pos+K)到anchors

    // 查找反向互补匹配
    rc_query ← getRC(query)
    FOR i从0到rc_query.length-K DO
        kmer ← rc_query.substr(i, K)
        IF kmer在kmerMap中 THEN
            q_start ← query.length - (i+K)
            q_end ← query.length - i
            FOR 每个r_pos IN kmerMap[kmer] DO
                添加Anchor(q_start, q_end, r_pos, r_pos+K, true)到anchors

    RETURN anchors

FUNCTION extendAnchors(anchors, query, ref):
    并行处理每个锚点:
        WHILE 可以向左扩展 AND 扩展长度 < MAX_EXTEND DO
            比较query和ref的相邻碱基
            IF 不匹配数 > MISMATCH_TOLERANCE THEN BREAK
            更新锚点边界

        WHILE 可以向右扩展 AND 扩展长度 < MAX_EXTEND DO
            比较query和ref的相邻碱基
            IF 不匹配数 > MISMATCH_TOLERANCE THEN BREAK
            更新锚点边界

    过滤掉长度 < MIN_ANCHOR_LENGTH 或相似度 < SIMILARITY_THRESHOLD的锚点

FUNCTION findOptimalChain(anchors):
    按q_start排序anchors

    // 动态规划找最优链
    dp ← 数组[anchors.size()]初始化为各锚点分数
    prev ← 数组[anchors.size()]初始化为-1

    FOR i从0到anchors.size()-1 DO
        FOR j从0到i-1 DO
            IF anchors[j]不与anchors[i]重叠 THEN
                gap_penalty ← calculateGapPenalty(anchors[j], anchors[i])
                new_score ← dp[j] + anchors[i].score - gap_penalty
                IF new_score > dp[i] THEN
                    dp[i] ← new_score
                    prev[i] ← j

    // 回溯找最优链
    max_idx ← dp中最大值的索引
    chain ← 空列表
```

```
    WHILE max_idx ≠ -1 DO
        将anchors[max_idx]添加到chain
        max_idx ← prev[max_idx]

    RETURN 反转后的chain

FUNCTION processGaps(chain, query, ref):
    regions ← 空列表

    // 处理第一个锚点前的区域
    IF 第一个锚点的q_start > 0 OR r_start > 0 THEN
        添加(0, q_start, 0, r_start)到regions

    // 添加锚点区域
    FOR 每个锚点 IN chain DO
        添加(q_start, q_end, r_start, r_end)到regions

        // 处理锚点间的gap
        IF 不是最后一个锚点 THEN
            q_gap_start ← 当前锚点.q_end
            q_gap_end ← 下一个锚点.q_start
            r_gap_start ← 当前锚点.r_end
            r_gap_end ← 下一个锚点.r_start
            IF q_gap_start < q_gap_end OR r_gap_start < r_gap_end THEN
                添加(q_gap_start, q_gap_end, r_gap_start, r_gap_end)到regions

    // 处理最后一个锚点后的区域
    IF 最后一个锚点.q_end < query.length OR r_end < ref.length THEN
        添加(q_end, query.length, r_end, ref.length)到regions

    RETURN regions
```

- 辅助函数伪代码

```
FUNCTION selectKmerSize(length):
    IF length < 500 THEN RETURN 10
    IF length < 2000 THEN RETURN 12
    IF length < 10000 THEN RETURN 14
    IF length < 50000 THEN RETURN 16
    IF length < 200000 THEN RETURN 18
    RETURN 20

FUNCTION getRC(s):
    rc ← 空字符串
    FOR 从s末尾到开头每个碱基b DO
        rc ← rc + b的互补碱基
    RETURN rc

FUNCTION calculateGapPenalty(a1, a2):
    q_gap ← a2.q_start - a1.q_end
    r_gap ← a2.is_rc ? (a1.r_start - a2.r_end) : (a2.r_start - a1.r_end)
    gap_diff ← |q_gap - r_gap|
```

```
    RETURN GAP_PENALTY_BASE + gap_diff * GAP_PENALTY_SCALE * (1 + log2(1 +
gap_diff))
```

复杂度分析

1. 主要组成部分

算法主要由以下几个关键部分组成：

k-mer索引构建

锚点查找

锚点扩展

锚点链构建

二级锚点查找

2. 各部分的复杂度分析

2.1 k-mer索引构建 (buildKmerIndex)

时间复杂度: O(L/K * T)

L是参考序列长度

K是k-mer大小

T是线程数

并行处理将参考序列分成T块，每块需要O(L/K)时间构建局部索引

空间复杂度: O(L)

存储所有k-mer及其位置需要与参考序列长度线性相关的空间

2.2 锚点查找 (findAnchors)

时间复杂度: O(Q * K + M)

Q是查询序列长度

K是k-mer大小

M是匹配的k-mer总数

需要遍历查询序列两次(正向和反向互补)

空间复杂度: O(M)

存储所有锚点

2.3 锚点扩展 (extendAnchors)

时间复杂度: O(A * E)

A是锚点数量

E是最大扩展长度(MAX_EXTEND)

每个锚点最多扩展E个碱基

空间复杂度: O(1)

原地修改锚点信息

2.4 锚点链构建 (findOptimalChain)

时间复杂度: $O(A^2)$

A是锚点数量

使用动态规划方法，对于每个锚点需要检查之前所有锚点

空间复杂度: O(A)

需要存储dp数组和前驱指针

2.5 二级锚点查找 (findSecondaryAnchors)

时间复杂度: O(U * K' + M')

U是未覆盖区域总长度

K'是二级k-mer大小

M'是二级匹配的k-mer总数

空间复杂度: O(M')

存储二级锚点

3. 整体复杂度

3.1 最坏情况时间复杂度

假设参考序列长度L，查询序列长度Q

最坏情况下(当所有k-mer都匹配):

k-mer索引构建: O(L)

锚点查找: O(QL) (因为可能有O(QL)个匹配)

锚点扩展: O(QLE)

锚点链构建: $O(Q^2*L^2)$

二级锚点查找: O(Q*L)

总最坏时间复杂度: $O(Q^2*L^2)$

3.2 平均情况时间复杂度

在实际生物序列中，k-mer匹配是稀疏的:

假设平均每个k-mer有C个匹配(C通常很小)

锚点数量A ≈ Q*C

各步骤复杂度:

k-mer索引构建: O(L)

锚点查找: O(Q + Q*C)

锚点扩展: O(QCE)

锚点链构建: $O(Q^2*C^2)$

二级锚点查找: O(Q + Q*C')

平均时间复杂度: $O(L + Q^2*C^2)$

3.3 空间复杂度

主要空间消耗:

k-mer索引: O(L)

锚点存储: O(Q*C)

动态规划表: O(Q*C)

总空间复杂度: O(L + Q*C)

运行结果

测试一

[(0,416,0,416), (416,832,416,832), (832,1248,832,1248), (1248,1664,1248,1664), (1664,2080,1664,2080), (2080,2496,2080,2496), (2496,2912,2496,2912), (2912,3328,2912,3328), (3328,3744,3328,3744), (3744,4160,3744,4160), (4160,4576,4160,4576), (4576,4992,4576,4992), (4992,5408,4992,5408), (5408,5824,5408,5824), (5824,6240,5824,6240), (6240,6656,6240,6656), (6656,6987,6656,6987), (6987,7000,6987,7001), (7000,7217,7001,7218), (7217,7219,7218,7221), (7219,7269,7221,7271), (7269,7277,7271,7275), (7277,7352,7275,7350), (7352,7357,7350,7355), (7357,7402,7355,7400), (7402,7444,7400,7441), (7444,7524,7441,7521), (7524,7556,7521,7555), (7556,7614,7555,7613), (7614,7679,7613,7672), (7679,7785,7672,7778), (7785,7786,7778,7779), (7786,7864,7779,7857), (7864,7895,7857,7888), (7895,7974,7888,7967), (7974,7995,7967,7988), (7995,8088,7988,8081), (8088,8097,8081,8090), (8097,8166,8090,8159), (8166,8167,8159,8160), (8167,8210,8160,8203), (8210,21625,8203,21619), (21625,21668,21619,21662), (21668,21669,21662,21663),

(21669,21738,21663,21732), (21738,21747,21732,21741), (21747,21840,21741,21834),
(21840,21861,21834,21855), (21861,21940,21855,21934), (21940,21963,21934,21957),
(21963,22049,21957,22043), (22049,22050,22043,22044), (22050,22156,22044,22150),
(22156,22215,22150,22215), (22215,22273,22215,22273), (22273,22328,22273,22326),
(22328,22391,22326,22389), (22391,22428,22389,22427), (22428,22473,22427,22472),
(22473,22478,22472,22477), (22478,22554,22477,22553), (22554,22557,22553,22560),
(22557,22607,22560,22610), (22607,22610,22610,22612), (22610,22827,22612,22829),
(22827,22841,22829,22842), (22841,23183,22842,23184), (23183,23599,23184,23600),
(23599,24015,23600,24016), (24015,24431,24016,24432), (24431,24847,24432,24848),
(24847,25263,24848,25264), (25263,25679,25264,25680), (25679,26095,25680,26096),
(26095,26511,26096,26512), (26511,26927,26512,26928), (26927,27343,26928,27344),
(27343,27759,27344,27760), (27759,28175,27760,28176), (28175,28591,28176,28592),
(28591,29007,28592,29008), (29007,29423,29008,29424), (29423,29829,29424,29830),
(29829,29845,29830,29830)]

测试二

[(0,301,0,301), (301,400,401,500), (400,501,498,599), (501,704,601,804), (704,804,704,804), (804,900,704,800),
(900,1000,699,799), (1000,1200,700,900), (1200,1300,899,999), (1300,1400,900,1000), (1400,1501,400,501),
(1501,1602,1001,1102), (1602,1700,1302,1400), (1700,1802,1200,1302), (1810,1900,1110,1200),
(1900,2002,1400,1503), (2018,2031,1607,1618), (2031,2044,852,863), (2056,2071,842,855), (2071,2082,975,987),
(2082,2096,1288,1301), (2096,2111,1084,1098), (2111,2122,1547,1558), (2122,2139,919,935),
(2139,2151,1159,1169), (2160,2178,1418,1435), (2178,2192,1245,1260), (2192,2209,1074,1089),
(2238,2248,1061,1071), (2248,2278,1064,1088), (2285,2295,1332,1343), (2299,2480,1499,1680)]