Lab1实验报告: 寻找样本DNA序列中的重复片段

洪运 2025/3/26

实验目的

详见实验文档。

实验内容

伪代码如下：

```
FUNCTION OptimalAlignment(query_seq, ref_seq):
    occ_list ← SearchOccurrences(query_seq, ref_seq)
    occ_table ← BuildOccurrenceTable(occ_list)   // 根据occ_list构建辅助信息表（可选）
    graph ← BuildGraphFromOccurrences(occ_list)

    // 初始化最短路径相关数据结构
    cost_arr[0...len(query_seq)] ← ∞
    cost_arr[0] ← 0
    min_heap ← EMPTY_HEAP
    PUSH(min_heap, (0, (ref_pos=0, query_pos=0, local_cost=0, reverse_flag=0)))
    visited ← EMPTY_SET
    prev_state ← EMPTY_MAP

    WHILE min_heap 不是空的:
        (curr_total_cost, (curr_r, curr_q, local_cost, rev_flag)) ← POP(min_heap)

        IF (curr_r, curr_q) 已经在 visited 中:
            CONTINUE
        ADD (curr_r, curr_q) 到 visited

        next_steps ← GetNextShifts(query_seq, ref_seq, graph, curr_r, curr_q)
        IF next_steps 为 None:
            CONTINUE

        FOR EACH (next_r, next_q, step_cost, next_rev) IN next_steps:
            IF next_q >= LENGTH(query_seq):  // 到达末尾状态
                route ← ReconstructRoute(prev_state, (curr_r, curr_q))
                CALL ShowAlignmentDetails(route, query_seq, graph)
                RETURN route
            END IF

            IF cost_arr[next_q] > cost_arr[curr_q] + step_cost:
                cost_arr[next_q] ← cost_arr[curr_q] + step_cost
                PUSH(min_heap, (cost_arr[next_q], (next_r, next_q, step_cost, next_rev)))
                prev_state[(next_r, next_q)] ← (curr_r, curr_q)
            END IF
        END FOR
    END WHILE

    PRINT "Failed to complete the alignment"
    RETURN FAILURE
END FUNCTION
```

```
FUNCTION SearchOccurrences(query_seq, ref_seq):
    occ_list ← EMPTY_LIST
    seg_dict ← EMPTY_MAP
    rev_ref ← ReverseComplement(ref_seq)

    // 枚举参考序列中的所有连续片段（正向与反向）
    FOR seg_length FROM 1 TO LENGTH(ref_seq):
        FOR r_start FROM 0 TO LENGTH(ref_seq) - seg_length:
            seg ← SUBSTRING(ref_seq, r_start, seg_length)
            rev_seg ← CorrespondingSegmentFrom(rev_ref, r_start, seg_length)
            ADD (r_start, seg_length, False) 到 seg_dict[seg]
            ADD (r_start, seg_length, True) 到 seg_dict[rev_seg]
        END FOR
    END FOR

    // 在查询序列中搜索与参考序列匹配的片段，统计重复次数
    FOR seg_length FROM 1 TO LENGTH(query_seq):
        FOR q_start FROM 0 TO LENGTH(query_seq) - seg_length:
            sub_query ← SUBSTRING(query_seq, q_start, seg_length)
            IF seg_dict 中有 sub_query:
                FOR EACH (r_start, r_seg_length, is_rev) IN seg_dict[sub_query]:
                    count ← 1
                    pos ← q_start + seg_length
                    WHILE pos + seg_length ≤ LENGTH(query_seq) AND SUBSTRING(query_seq, pos, seg
                        count ← count + 1
                        pos ← pos + seg_length
                    END WHILE
                    APPEND (r_start, q_start, seg_length, count, is_rev) 到 occ_list
                END FOR
            END IF
        END FOR
    END FOR

    RETURN occ_list
END FUNCTION


FUNCTION BuildGraphFromOccurrences(occurrence_list):
    graph ← NEW Graph()
    FOR EACH (r_start, q_start, seg_length, count, is_rev) IN occurrence_list:
        q_end ← q_start + seg_length * count
        // 边的权重为count，其元数据保存额外信息
```

```
        graph.ADD_EDGE(from=q_start, to=q_end, weight=count, meta={ref_start: r_start,
                                                                   seg_length: seg_length,
                                                                   count: count,
                                                                   is_rev: is_rev})
    END FOR
    RETURN graph
END FUNCTION


FUNCTION GetNextShifts(query_seq, ref_seq, graph, curr_r, curr_q):
    neighbors ← graph.NEIGHBORS(curr_q)
    shifts ← EMPTY_LIST

    IF neighbors 为空:
        IF curr_q < LENGTH(query_seq) AND curr_r < LENGTH(ref_seq) AND
           query_seq[curr_q] == ref_seq[curr_r]:
            APPEND (curr_r+1, curr_q+1, 0, 0) TO shifts
        ELSE:
            PRINT "Alignment failed at query position", curr_q, "reference position", curr_r
            RETURN None
        END IF
    ELSE:
        // 可选择普通步进
        IF curr_q < LENGTH(query_seq) AND curr_r < LENGTH(ref_seq) AND
           query_seq[curr_q] == ref_seq[curr_r]:
            APPEND (curr_r+1, curr_q+1, 0, 0) TO shifts
        END IF

        FOR EACH dest IN neighbors:
            weight ← graph.GET_WEIGHT(curr_q, dest)
            meta ← graph.GET_META(curr_q, dest)
            new_r ← curr_r
            IF meta.ref_start == curr_r:
                new_r ← curr_r + meta.seg_length
                weight ← weight - 1
            END IF
            APPEND (new_r, dest, weight, meta.is_rev) TO shifts
        END FOR
    END IF

    RETURN shifts
END FUNCTION
```

```
FUNCTION ShowAlignmentDetails(path, query_seq, graph):
    PRINT "Repeated mutations in the optimal path:"
    PRINT "Index   RefStart   QueryStart   SegLength   Repeats    Segment         Reverse"

    rep_index ← 0
    FOR i FROM 0 TO LENGTH(path)-2:
        (curr_r, curr_q) ← path[i]
        (next_r, next_q) ← path[i+1]
        IF (next_q - curr_q) > 1:
            IF graph.EDGE_EXISTS(curr_q, next_q):
                rep_index ← rep_index + 1
                meta ← graph.GET_META(curr_q, next_q)
                seg ← SUBSTRING(query_seq, curr_q, meta.seg_length)
                actual_repeat ← (next_q - curr_q) DIV meta.seg_length
                IF meta.ref_start == curr_r:
                    actual_repeat ← actual_repeat - 1
                END IF
                IF actual_repeat > 0:
                    PRINT rep_index, meta.ref_start, curr_q, meta.seg_length, actual_repeat, seg
                END IF
            ELSE:
                PRINT "Alignment from query pos", curr_q, "to", next_q
            END IF
        END IF
    END FOR
END FUNCTION
```

说明：

1. 函数 SearchOccurrences 扫描参考序列与查询序列，查找所有匹配的子串并统计重复次数。
2. BuildGraphFromOccurrences 根据搜索结果构造有向图，每条边的起始位置为查询序列位置，将重复发生的位置连接起来。
3. GetNextShifts 根据当前状态（curr_r, curr_q），利用图中的边得出下一步可能的状态。若无特殊边，则尝试逐字符匹配。
4. 主函数 OptimalAlignment 利用 Dijkstra（或A*类似）算法，在最小堆中扩展状态，直至遍历完查询序列。满足结束条件时，调用 ShowAlignmentDetails 输出重复变异详细信息。

时间复杂度分析：
在参考序列中，通过双层循环枚举所有长度从 1 到 n(n 为参考序列长度)的子串，时间复杂度约为 $O(n^2)$。
在查询序列中，同样使用双层循环查找所有可能的片段，时间复杂度约为 $O(m^2)$(m 为查询序列长度)。

虽然字典查找每次是均摊 O(1) 的，但枚举所有子串已经使这一部分最坏情况达到 $O(n^2 + m^2)$。
由于m >> n, 所以整体时间复杂度为 $O(m^2)$。

具体代码实现与文件结构如下：

```
--DNAMatch
    |--data
        |--query1.txt
        |--query2.txt
        |--reference1.txt
        |--reference2.txt
    --alignment.py
    --graph.py
    --main.py
    --utils.py
```

```python
# alignment.py

import heapq
from utils import reverse_complement
from graph import Graph

def search_occurrences(query_seq, ref_seq):
    q_len = len(query_seq)
    r_len = len(ref_seq)

    occurrences = []
    seg_dict = {}
    rev_ref = reverse_complement(ref_seq)

    for seg_length in range(1, r_len + 1):
        for r_start in range(r_len - seg_length + 1):
            seg = ref_seq[r_start:r_start + seg_length]
            rev_seg = rev_ref[r_len - (r_start + seg_length):r_len - r_start]
            seg_dict.setdefault(seg, []).append((r_start, seg_length, False))
            seg_dict.setdefault(rev_seg, []).append((r_start, seg_length, True))

    for seg_length in range(1, q_len + 1):
        for q_start in range(q_len - seg_length + 1):
            sub_query = query_seq[q_start:q_start + seg_length]
            if sub_query in seg_dict:
                for (r_start, r_seg_length, is_rev) in seg_dict[sub_query]:
                    count = 1
                    pos = q_start + seg_length
                    while pos + seg_length <= q_len and query_seq[pos:pos + seg_length] == sub_
                        count += 1
                        pos += seg_length
                    occurrences.append((r_start, q_start, seg_length, count, is_rev))

    return occurrences

def build_graph_from_occurrences(occurrence_list):
    graph = Graph()
    for r_start, q_start, seg_length, count, is_rev in occurrence_list:
        q_end = q_start + seg_length * count
        graph.add_edge(q_start, q_end, count, {'ref_start': r_start, 'seg_length': seg_length,
    return graph

def get_next_shifts(query_seq, ref_seq, graph, curr_r, curr_q):
```

```
    neighbors = graph.neighbors(curr_q)
    shifts = []
    if not neighbors:
        if curr_q < len(query_seq) and curr_r < len(ref_seq) and query_seq[curr_q] == ref_seq[cu
            shifts.append((curr_r + 1, curr_q + 1, 0, 0))
        else:
            print(f"Alignment failed at query position {curr_q}, reference position {curr_r}.")
            return None
    else:
        if curr_q < len(query_seq) and curr_r < len(ref_seq) and query_seq[curr_q] == ref_seq[cu
            shifts.append((curr_r + 1, curr_q + 1, 0, 0))
        for dest in neighbors:
            weight = graph.get_weight(curr_q, dest)
            meta = graph.get_meta(curr_q, dest)
            new_r = curr_r
            if meta['ref_start'] == curr_r:
                new_r += meta['seg_length']
                weight -= 1
            shifts.append((new_r, dest, weight, meta['is_rev']))
    return shifts

def show_alignment_details(path, query_seq, graph):
    print("\nRepeated mutations in the optimal path:")
    header = f"{'Index':<6} {'RefStart':<10} {'QueryStart':<12} {'SegLength':<10} {'Repeats':<8}
    print(header)
    rep_index = 0
    for i in range(len(path) - 1):
        curr_r, curr_q = path[i]
        next_r, next_q = path[i + 1]

        if next_q - curr_q > 1:
            if graph.edge_exists(curr_q, next_q):
                rep_index += 1
                meta = graph.get_meta(curr_q, next_q)
                seg = query_seq[curr_q:curr_q + meta['seg_length']]
                actual_repeat = (next_q - curr_q) // meta['seg_length']
                if meta['ref_start'] == curr_r:
                    actual_repeat -= 1
                if actual_repeat > 0:
                    print(f"{rep_index:<6} {meta['ref_start']:<10} {curr_q:<12} {meta['seg_lengt
            else:
                print(f"Alignment from query pos {curr_q} to {next_q}.")
    return
```

```
def print_occurrence_list(occurrences):
    if not occurrences:
        print("No repeat mutations detected")
        return
    print(f"{'Index':<6} {'RefStart':<10} {'QueryStart':<12} {'SegLength':<10} {'RepeatCount':<1
    for i, (r_start, q_start, seg_length, count, is_rev) in enumerate(occurrences, 1):
        print(f"{i:<6} {r_start:<10} {q_start:<12} {seg_length:<10} {count:<12} {is_rev!s:<10}"

def optimal_alignment(query_seq, ref_seq):
    occ_list = search_occurrences(query_seq, ref_seq)
    occ_table = {}
    for occ in occ_list:
        r_start, q_start, seg_length, count, is_rev = occ
        occ_table.setdefault(q_start, {}).setdefault(r_start, []).append({
            'ref_start': r_start,
            'seg_length': seg_length,
            'count': count,
            'is_rev': is_rev
        })

    graph = build_graph_from_occurrences(occ_list)
    q_len = len(query_seq)
    heap = []
    cost_arr = [float('inf')] * (q_len + 1)
    cost_arr[0] = 0
    visited = set()
    prev_state = {}
    heapq.heappush(heap, (0, (0, 0, 0, 0)))

    while heap:
        curr_total_cost, state = heapq.heappop(heap)
        curr_r, curr_q, local_cost, rev_flag = state
        if (curr_r, curr_q) in visited:
            continue
        visited.add((curr_r, curr_q))

        next_steps = get_next_shifts(query_seq, ref_seq, graph, curr_r, curr_q)
        if next_steps is None:
            continue
        for next_r, next_q, step_cost, next_rev in next_steps:
            if next_q >= q_len:
                route = []
```

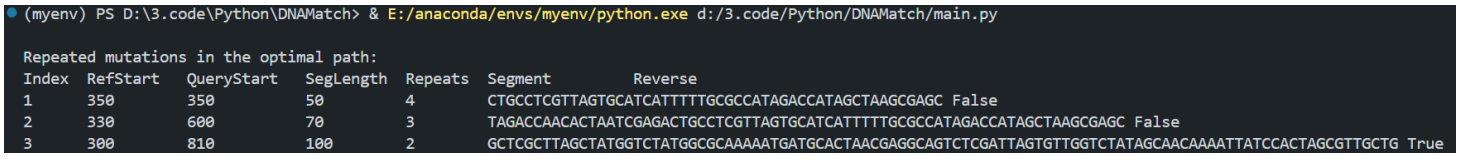
```
                curr_state = (curr_r, curr_q)
                while curr_state != (0, 0):
                    route.append(curr_state)
                    curr_state = prev_state.get(curr_state, (0,0))
                route.reverse()
                show_alignment_details(route, query_seq, graph)
                return route
            if cost_arr[next_q] > cost_arr[curr_q] + step_cost:
                cost_arr[next_q] = cost_arr[curr_q] + step_cost
                heapq.heappush(heap, (cost_arr[next_q], (next_r, next_q, step_cost, next_rev)))
                prev_state[(next_r, next_q)] = (curr_r, curr_q)

    print("Failed to complete the alignment")



# graph.py

class Graph:
    def __init__(self):
        self.adjacency = {}

    def add_edge(self, src, dest, weight, meta):
        self.adjacency.setdefault(src, {})[dest] = {'weight': weight, 'meta': meta}

    def edge_exists(self, src, dest):
        return src in self.adjacency and dest in self.adjacency[src]

    def neighbors(self, src):
        if src not in self.adjacency:
            return []
        return list(self.adjacency[src].keys())

    def get_meta(self, src, dest):
        return self.adjacency[src][dest]['meta']

    def get_weight(self, src, dest):
        return self.adjacency[src][dest]['weight']
```

```python
# utils.py

def reverse_complement(dna_seq):
    complement = {'A': 'T', 'T': 'A', 'C': 'G', 'G': 'C'}
    return ''.join(complement.get(nuc, nuc) for nuc in reversed(dna_seq))

def read_seq(filename):
    with open(filename, 'r') as f:
        return f.read().strip()
```

```python
# main.py

from utils import read_seq
from alignment import optimal_alignment

def main():
    query = read_seq("data/query1.txt")
    reference = read_seq("data/reference1.txt")
    optimal_alignment(query, reference)

if __name__ == "__main__":
    main()
```

实验结果

如图:



修改main.py中的文件名，运行不同的查询序列与参考序列，得到的结果如下：

```
Repeated mutations in the optimal path:
Index  RefStart  QueryStart  SegLength  Repeats  Segment          Reverse
1      1228      76          6          1        GCGTGT           False
2      82        82          49         1        TTCTACGAGCCAGGTGCTCATGCCTAAGTTCTGCGCCTTCGCTGTCACT False
3      132       132         16         1        GGAAATACTGTAATGG False
4      247       148         6          1        ATCATC           True
5      156       156         34         1        TAGGTTATGCGCCTTCGGGGTCACTTCAACATAC False
6      1232      190         6          1        TGTAAC           True
7      396       196         6          1        GGATCG           False
8      252       202         9          1        TGCCTAGGT        False
9      1617      211         8          1        TTTGCGTA         False
10     219       219         14         1        TTCGCTGTCATTTC   False
11     1478      233         8          1        GAATTACA         False
12     241       241         29         1        CCAATGGATGATGCCTAGGTTCTGTGCCT False
13     1019      270         7          1        CCGCTGA          True
14     654       277         6          1        CGCATC           False
15     283       283         13         1        GAAATACTTTAAC    False
16     396       296         8          1        GGATCGCG         False
17     1609      304         5          1        TCCGA            False
18     409       309         22         1        GTAACAACTTCGTACTGTTATA False
19     252       331         6          1        TAGGCA           True
20     437       337         10         1        ATCAGAATAC       False
21     916       347         5          1        CCATG            True
22     452       352         48         1        CCTCGGGGAGGTAACCTATATTCACGTCGCAAGTTTCGATCTACAGTA False
23     591       400         9          1        CTGTAGGTA        True
24     570       410         20         1        ATCTTTTGGTGTCATATGAG True
25     1622      431         5          1        GTACT            False
26     1685      436         4          1        GAAC             False
27     542       440         18         1        TTGACGACCGCGACCATG True
28     918       458         5          1        TGGAT            False
29     511       463         26         1        GCGCTTCTTGACTTAAAATCGCGGCA True
30     421       489         7          1        AACAGTA          True
31     1578      496         5          1        AGCAT            True
32     953       501         6          1        CCGTGA           False
33     607       507         52         1        AGCTCGACTTTTGTGCTCTAGGGCACGACGGGTGGCGTTTGCTCCCGCGCAT False
34     29        559         8          1        CTCGAGTT         True
35     668       568         16         1        TAAGCTCTATGGCACA False
36     834       584         9          1        ACGGGTGGC        False
37     508       593         8          1        GTTTGCCG         False
38     1610      601         4          1        CCGA             False
39     705       605         83         1        GCAGCTCGACTTTTGTGCTCTAGGGCACGGCGGGTGGCGTTTGCCCTCGCCCAGCTTGACTTTTGTGCTCTAGGGCACGACGG False
40     1554      688         6          1        GTGGCC           False
41     694       694         13         1        TTTGCCCCCGCGC    False
42     1605      707         5          1        AGCTG            True
43     712       712         37         1        GACTTTTGTGCTCTAGGGCACGGCGGGTGGCGTTTGC False
44     1531      749         5          1        CCTCC            False
45     754       754         9          1        CCCAGCTTG        False
46     1595      763         5          1        ACTAC            True
```

由于显示问题未能截全。

GitHub仓库地址：

Algorithm-H-2025-Fudan