CS238: Homework 2

Sakshar Chakravarty (862253039)

February 10, 2022

Q. 1

This can be solved very easily by extending the solution for linear genome. The basic change is how we define the breakpoints. Apart from that the solution is almost the same.

- **Breakpoint:** A breakpoint in a circular permutation can be defined at a position, where two adjacent numbers c_i and c_{i+1} in the permutation, are not consecutive with the only exception for 1 and n which are considered consecutive in a circular permutation that is if $i = n$, then $i + 1 = 1$.
- **Strip:** It is the same as it was for linear permutation. However, we do not need to tag a strip based on the ascending or descending order in case of circular permutation.

The algorithm 1 always reverses strip/s which are substring/s split by breakpoints. So, a reversal operation cannot introduce a new breakpoint. Repeatedly, it reduces breakpoints in the permutation and finally produces the identity circular permutation. If we look into steps 2 and 3 of algorithm 1, we can see that

Algorithm 1 Sorting by Reversal for Circular Genome

1. Pick a $c_i < n$ in such a way that c_i and $c_i + 1$ are not in the same strip.
2. Perform a single reversal to make both the strips containing c_i and $c_i + 1$ adjacent.
3. If step 2 cannot put c_i and $c_i + 1$ side by side, reverse the strip with $c_i + 1$ to make them adjacent.
4. Repeat steps $1 - 3$ until there is no more breakpoints.

we are merging two strips into one and it takes either one or two reversals to do this and subsequently reduce one breakpoint.

Approximation Ratio: Removing a single breakpoint may take at most 2 reversals. So, at most the algorithm may require $2 * br(\pi)$ reversals, if $br(\pi)$ be the number of breakpoints in the given circular genome π. The best solution can remove 2 breakpoints in a single reversal operation. So, following the optimal solution, identity permutation can be obtained by $\frac{br(\pi)}{2}$ reversals to the least. Therefore, algorithm 1 has an approximation ratio of $\frac{2*br(\pi)}{br(\pi)/2} = 4$.

Q. 2

First, let's look into an $O(n^2)$ algorithm to find the breakpoint distance between two given permutations π_1 and π_2 consisting of integers from 1 to n.

Algorithm 2 Pairwise Breakpoint Distance Problem

```
Add 0 and n + 1 at the beginning and the end of both π1 and π2
dist ← 0
for i from 0 to n do
    for j from 0 to n + 1 do
        if π1[i] = π2[j] then
            if j = 0 and π1[i + 1] ≠ π2[j + 1] then
                dist ← dist + 1
            else if j = n + 1 and π1[i + 1] ≠ π2[j − 1] then
                dist ← dist + 1
            else if not (π1[i + 1] = π2[j + 1] or π1[i + 1] = π2[j − 1]) then
                dist ← dist + 1
            end if
        end if
    end for
end for
return dist
```

Now, let's design a greedy algorithm to solve the multiple breakpoint distance problem. The basic idea is similar to Center-Star algorithm. The overall complexity of this greedy approach is $O(k^2n^2)$ where k is the number of permutations $\pi_1, \pi_2, ..., \pi_k$ each of length n.

Algorithm 3 Greedy Multiple Breakpoint Distance Problem

```
minDist, σ' ← ∞, null
for i from 1 to k do
    currentDist ← 0
    for j from 1 to k do
        currentDist ← currentDist + br(πi, πj)
    end for
    if currentDist < minDist then
        minDist, σ' ← currentDist, πi
    end if
end for
return σ'
```

Approximation Ratio: Suppose, σ be the optimal ancestral permutation and π_g be the greedily obtained permutation. $d(\sigma) = \sum_{1 \leq j \leq k} br(\sigma, \pi_j)$ be the optimal breakpoint distance and $d(\pi_g) = \sum_{1 \leq j \leq k} br(\pi_g, \pi_j)$ be the sum of the breakpoint distances between the greedy solution and the rest of the permutations. It

is evident from the calculation of pairwise breakpoint distance that it follows the triangle inequality property. Therefore, $br(\pi_m, \pi_n) \leq br(\pi_m, \pi_p) + br(\pi_p, \pi_n)$. From this, we get,

$$d(\pi_g) = \sum_{i \neq j} br(\pi_i, \pi_j) \leq \sum_{i \neq j} (br(\pi_i, \pi_g) + br(\pi_g, \pi_j)) = 2(k-1) \sum_{i \neq g} br(\pi_g, \pi_i) \quad (1)$$

But for the optimal ancestral permutation σ, we get,

$$d(\sigma) = \sum_{i} \sum_{i \neq j} br(\pi_i, pi_j) = k \sum_{i \neq j} br(\pi_i, \pi_j) \quad (2)$$

Therefore, by dividing eq. 1 with eq. 2, we get,

$$\frac{d(\pi_g)}{d(\sigma)} \leq \frac{2(k-1)}{k} \approx 2 \quad (3)$$

Q. 3

Fitting alignment problem is kind of a fusion of global and local alignment. Suppose, we are given two sequences a and b of length m and n where $m < n$. The goal of the fitting alignment is to find out the substring of b with which the whole of the sequence a aligns the best. So, the optimal alignment may start and end any where in b but it should cover the whole of a. The initialization and recurrence steps are as follows:

Initialization: We declare a matrix S of size $(m+1) \times (n+1)$. Also, we are given a scoring scheme δ for matches, mismatches, and indels. Then we fill out the first row and the first column in the following manner:

- First row: $S[0][j] = 0$ for $0 \leq j \leq n$. Since the alignment can start anywhere in the first row.
- First column: $S[i][0] = S[i-1][0] + \delta(a_i, -)$ for $1 \leq i \leq m$

Recurrence relation: For $1 \leq i \leq m$ and $1 \leq j \leq n$,

$$S[i][j] = max \begin{cases} S[i-1][j] + \delta(a_i, -) \\ S[i][j-1] + \delta(-, b_j) \\ S[i-1][j-1] + \delta(a_i, b_j) \end{cases}$$

Optimal score: The optimal fitting alignment score will be found at a cell in the last row with the maximum value. Since the alignment can end anywhere in the last row.

$$opt = max\{S[m][:]\}$$

Backtracing: We start from the cell with maximum value in the last row. Then recursively follow the path along the pointers which were stored while populating the scoring matrix until we reach the first row. The path obtained from backtracing will give us the optimal fitting alignment.

Complexity: The algorithm fills out a matrix of order $O(mn)$ using a dynamic programming approach. Therefore, both time and space complexity of this algorithm is $O(mn)$.

Q. 4

Since the weights are same, we can consider each exon, e_i as a pair like (s_i, t_i) where s_i indicates the starting and t_i indicates the ending positions/co-ordinates in the genome sequence. We need to find out the maximal set of non-overlapping exons. The greedy approach for solving this problem is similar to task scheduling algorithm. It is as follows:

1. Sort the given set of exons, E in ascending order of their ending positions. So, in this sorted order $t_1 \leq t_2 \leq ... \leq t_n$.
2. We initialize an empty set R to hold the selected exons.
3. We extract the first exon from the current sorted set of exons E and put it into the result set R.
4. We remove the exons after it with starting positions smaller than the ending position of the exon picked in step 3.
5. Repeat steps $3 - 4$ until the ordered set of exons E becomes empty.
6. Return R which is the maximal set of non-overlapping exons.

Analysis: If the given set contains n exons, the first step that is sorting them requires $O(n \log n)$. Afterwards, we basically scan through the sorted order once in a loop. So, building the maximal set R from the sorted order E takes $O(n)$. Therefore, the overall run-time of the algorithm is $O(n \log n)$.