CAP-6515 FINAL

1 Solution to Question No: 1

1.1 Problem Statement

In a splice alignment algorithm, Given a known mRNA T and a genome G, we have to find the set of exons in G so that the alignment score of the exon chain is maximum. Here, given input are a Genomic sequences G, target mRNA sequence T, and a set of putative exons B. One limitation of this algorithm is that, in order to maximize the score it may concatenate many short exons, there is no limit to the number of exons. We have to modify the splicing algorithm in such a way that there can be maximum k exons in the exon chain. The output of the algorithm will be the exon chain with the maximal alignment score having no more than k exons.

1.2 Modified Recursive Equation

We'll need an additional array called Count[B], it's size will be equal to B (the number of exon blocks considered in the genome in total). This will be used to store the number of blocks that have been considered till now to reach the maximum alignment score for the current block. We have to initialize count of 1st block, Count[B_1] to 1.

For example: if current block is B_j and its starting position score is coming from block B_i where $i < j$, then Count[B_j] = Count[B_i] + 1. If the maximum score of the current block B_j is not coming from any of the previous blocks, then Count[B_j] = 1 (first block of the chain). As the number of exons in the chain cannot be more than k, before calculating the maximum score for any current block B_j's starting index, we have to give a check if Count[B_i] $< k$ (Here, B_i is a block considered before B_j). If Count[B_i] $\geq k$, then even if the alignment score coming from that chain was maximum, still we have to discard that possibility and look for the next one.
Suppose, Genome sequence, G = $G_1G_2....G_i......$
mRNA sequence, T = $T_1T_2....T_j......$
Set of putative exons B = $B_1, B_2....$ the modified recursive equation is given below:

Initialization: Count[B_1] = 1

If i is not the starting vertex of Block B then,

$$S(i,j,B) = \max\begin{cases} S(i-1,j,B) - \textit{indel penalty} \\ S(i,j-1,B) - \textit{indel penalty} \\ S(i-1,j-1,B) + \delta(g_i,t_j) \end{cases}$$

If i is the starting vertex of Block B then,

$$S(i,j,B) = \max\begin{cases} S(i,j-1,B) - \textit{indel penalty} \\ \max\{S(end(B'),j,B') - \textit{indel penalty}\ s.t.\ B' \in \textit{preceding blocks of } B \textit{ and } \mathbf{Count}[B'] < k\} \\ \max\{S(end(B'),j-1,B') + \delta(g_i,t_j)\ s.t.\ B' \in \textit{preceding blocks of } B \textit{ and } \mathbf{Count}[B'] < k\} \end{cases}$$

$$\mathbf{Count}(B) = \begin{cases} \mathbf{Count}(B') + 1; & \textit{if value of } S(i,j,B) \textit{ comes from a preceding block } B' \\ \mathbf{1}; & \textit{otherwise} \end{cases}$$

After computing the three-dimensional table S(i, j, B), the score of the optimal spliced alignment is:

$$\max_{\text{all blocks } B} S(end(B), length(T), B)$$

2 Solution to Question No: 2

2.1 Problem Statement

In a DNA sequence shuffling problem, for a constant size of k, the k-mers are shuffled randomly such that the frequency of the k-mers remain unchanged as the input sequence. If k=1 then this can be easily done using a swapping algorithm. But swapping algorithm isn't efficient enough for k>1. Here, we have to design a linear-time sequence shuffling algorithm where the frequency of each k-mer in the shuffled sequence should be same as the input sequence. Also, the algorithm has to be designed in such a way so that it outputs a random shuffled sequence every time among the multiple possible shuffled sequences.

2.2 Proposed Algorithm

Suppose the given input sequence is S and it's length is N. Here the value of k is user-input and has to remain constant throughout the algorithm. Suppose E (spectrum) is the set of all possible k-mers in the given string S. If some k-mer occurs multiple times then E contains that k-mer multiple times (to keep track of the frequency of k-mers). Now, we have to build a De Bruijn graph G where all the k-mers in set E will be represented as edges. If there are multiple same k-mers then there will be multiple edges for them. Suppose V represents the set of vertices in the graph and E is the set of edges. For each edge $S_iS_{i+1}...S_{i+k-1}$ there will be two vertices added in vertex set V, vertex $S_iS_{i+1}...S_{i+k-2}$ and vertex $S_{i+1}S_{i+2}...S_{i+k-1}$. No same vertex will be added twice to the vertex set V. Once, we have the set V and E, graph G can be build in O(V+E) time.

A directed graph has Euler path if there are at most two odd degree (in-degree != out-degree) vertices. De Bruijn graph built from a sequence always has a Euler path. Suppose, 'Start' represents a set of vertices from which the Euler path can start. If all the vertices have equal no. of in and out degree, then that graph is also an Eulerian graph and Euler path can start from any of them. So, 'Start' = V. Otherwise, there will be one vertex v1 for which in-degree + 1 = out-degree and another vertex v2 for which in-degree = out-degree + 1; such graphs are known as Semi-Euler graph. Euler path will always start from vertex v1, so 'Start' = v1. While finding Euler path from graph G, we have to select any one vertex from 'Start' set.

We have to keep two sets named Current_Vertices and Path. Current_Vertices will keep track of the vertices that are being visited right now and that might still have unvisited edges. Whenever the algorithm will find a vertex with all visited edges, that will be added to the set Path. When all edges have been visited, remaining vertices in Current_Vertices will be added to Path in reverse order as they also don't have anymore unvisited edges. Then the vertices of Path will be added to set Eulerian_path in reverse order. This Eulerian_path represents the actual sequence of vertices in the Euler path for graph G. The shuffled sequence S' can be subtracted from Eulerian_path set (using the sequence of edges visited). To make the whole process random for generating random shuffled sequences, we will select an edge randomly from the current vertex using rand() function. Here, Rand_Max will be equal to the number of remaining outgoing edges from the current vertex.

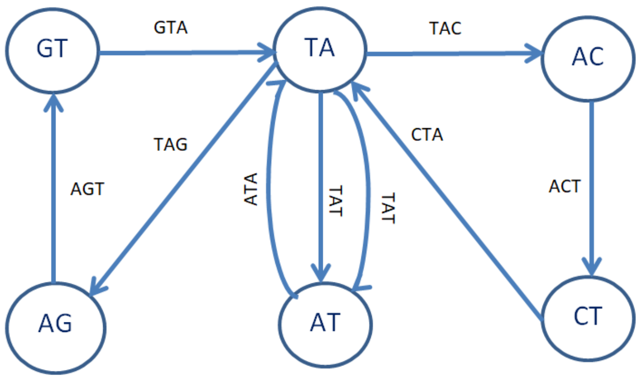
The algorithm for generating one random shuffled sequence in linear-time is given below:

Algorithm 1 Linear-time Random Sequence Shuffling Algorithm

```
1:  N ← length(S)
2:  k ← User-input                                        ▷ Generates a value between 1 and N
3:  E ← All possible set of k-mers                        ▷ k-mer composition
4:  Start ← NULL                                          ▷ Set of Starting Vertices for possible Euler Path
5:  Current_Vertices ← NULL
6:  Path ← NULL
7:
8:  Form a De Bruijn Graph G for edges E with V vertices  ▷ Each k-mers will be edge of the graph
9:  Check if graph G is Euler (all vertexes balanced) or Semi-Euler(at most two vertices unbalanced)
10: if G Eulerain then
11:     Start = V
12: else
13:     Start = { v ∈ V : v has in-degree + 1 = out-degree } ▷ In Semi-Euler graph, Eulerian path always starts with the vertex
                                                            that has out-degree > in-degree and out-degree= in-degree+1
14: Select a vertex v1 randomly from Start
15: Current_Vertices = Current_Vertices + v1              ▷ Add vertex v1 as current
16:
17: while E != NULL do
18:     Rand_Max = number of Out Edges from v1
19:     if Rand_Max = 0 then                              ▷ No Out-edge from v1 left
20:         Path = Path + v1                              ▷ Add vertex v1 to Path
21:         Current_Vertices = Current_Vertices - v1      ▷ Remove vertex v1 from current
22:         if Current_Vertices = NULL then
23:             Break                                     ▷ No more edges to visit
24:         else
25:             v1 = top(Current_Path)
26:     else
27:         Selected_edge = rand() % Rand_Max +1          ▷ Selects one Out-edge from v1 randomly
28:         E = E - Selected_edge
29:         v1 = Vertex for which Selected_edge is In-Edge ▷ Next Out-edge of this vertex will be considered
30:         Current_Vertices = Current_Vertices + v1      ▷ Add vertex v1 as current
31:
32: if Current_Vertices != NULL then
33:     Path = Path + reverse(Current_Vertices)           ▷ Add elements of Current_Vertices to Path in reverse order
34:
35: Eulerian_Path = reverse(Path)     ▷ Visiting the vertices of Path in reverse order will give the Eulerian path in graph G
36: S' = shuffled sequence found from visiting the edges of Eulerian_Path
37: return S'
```

2.3 Correctness Proof With Example

Suppose S = TACTATAGTAT
E = TAC, ACT, CTA, TAT, ATA, TAG, AGT, GTA, TAT
V= TA, AC, CT, AT, AG, GT
De Bruijn Graph is given below:



The graph G is semi-eular, as vertex TA and AT has odd degree. So, Start = TA (Out-degree = In-degree + 1)

Now, v1 = TA
Current_Vertices = {TA}

While Loop Starts:

Iteration 1:
Random_Max = 4 (as v1 = TA)
Suppose randomly Selected_Edge = 3 (TAT in the middle)
E = {TAC, ACT, CTA, TAT, ATA, TAG, AGT, GTA}
v1 = AT
Current_Vertices = {TA, AT}

Iteration 2:
Random_Max = 1 (as v1 = AT)
Selected_Edge = 1 (ATA)
E = {TAC, ACT, CTA, TAT, TAG, AGT, GTA}
v1 = TA
Current_Vertices = {TA, AT, TA}

Iteration 3:
Random_Max = 3 (as v1 = TA)
Selected_Edge = 2 (TAT in the right)
E = {TAC, ACT, CTA, TAG, AGT, GTA}
v1 = AT
Current_Vertices = {TA, AT, TA, AT}

Iteration 4:
Random_Max = 0 (as v1 = AT)
Path = {AT}
Current_Vertices = {TA, AT, TA}
v1 = TA

Iteration 5:
Random_Max = 2 (as v1 = TA)
Selected_Edge = 4 (TAG)
E = {TAC, ACT, CTA, AGT, GTA}
v1 = AG
Current_Vertices = {TA, AT, TA, AG}

Iteration 6:
Random_Max = 1 (as v1 = AG)
Selected_Edge = 1 (AGT)
E = {TAC, ACT, CTA, GTA}
v1 = GT
Current_Vertices = {TA, AT, TA, AG, GT}

Iteration 7:
Random_Max = 1 (as v1 = GT)
Selected_Edge = 1 (GTA)
E = {TAC, ACT, CTA}
v1 = TA
Current_Vertices = {TA, AT, TA, AG, GT, TA}

Iteration 8:
Random_Max = 1 (as v1 = TA)
Selected_Edge = 1 (TAC)
E = {ACT, CTA}
v1 = AC
Current_Vertices = {TA, AT, TA, AG, GT, TA, AC}

Iteration 9:
Random_Max = 1 (as v1 = AC)
Selected_Edge = 1 (ACT)
E = {CTA}
v1 = CT
Current_Vertices = {TA, AT, TA, AG, GT, TA, AC, CT}

Iteration 10:
Random_Max = 1 (as v1 = CT)
Selected_Edge = 1 (CTA)
E = { }
v1 = TA
Current_Vertices = {TA, AT, TA, AG, GT, TA, AC, CT, TA}

End of While loop.
Now Path = Path + reverse(Current_Vertices) = {AT, TA, CT, AC, TA, GT, AG, TA, AT, TA}
Now Eulerian_path = {TA, AT, TA, AG, GT, TA, AC, CT, TA, AT}
Edges traversed = TAT, ATA, TAG, AGT, GTA, TAC, ACT, CTA, TAT

Output shuffled sequences, S' = TATAGTACTAT

2.4 Time Complexity

Here, forming the graph takes O(V+E) time. Finding the Eulerian path from graph G using Hierholzer's algorithm requires O(E). So, the runtime is linear.

3 Acknowledgment

To understand the concept better, took help from the following sites.

1) http://www.graph-magics.com/articles/euler.php
2) http://what-when-how.com/bioinformatics/spliced-alignment-bioinformatics/