Problem 1:

(A) In order to uncompress the data, you need to send the compressed data, the initial symbol order and the probability model of the symbols.

(B) Suppose that the initial symbol order is \{A,B,C,D,E\}. Using this initial symbol order to encode ABCAB using move-o-front coding, we will first get

\{0,1,2,2,2,1,1,2,2,2,1,1,2,2,2,…\}

Repeating pattern

To calculate the asymptotically how many arithmetic coding use to store this string, for the above string that is long enough, we have p(0)=0, p(1)=0.4, p(2)=0.6.

Note that \(\lim_{n \to \infty} p \log_2 \frac{1}{p} = 0\), we only need to calculate how much space averagely it needs to store 1 and 2 and calculate the average weight of them. It can be calculated as:

\[
0.4 \times \log_2 \frac{1}{0.4} + 0.6 \times \log_2 \frac{1}{0.6} = 0.529 + 0.442 = 0.97
\]

Thus, asymptotically 0.97 bits-per-symbol will be the move-to-front heuristic use to store the string.

(C) Suppose that the initial symbol order is \{A,B,C,D,E\}. Using this initial symbol order to encode ABCDE using move-o-front coding, we will first get

\{0,1,2,3,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,…\}

Repeating pattern

In this case, only 4 has non-zero probability if the length of the code is infinite. And it’s probability p(4)=1. For others, p(other)=0.

Thus, asymptotically the move-to-front heuristic will use \(1 \times \log_2 \frac{1}{1} = 0\) bits-per-symbol to store the string.

(D) Yes, it will. That’s because for different message types which have different pattern, after certain iterations, move-to-front coding will adapt the symbol ordering list to “match”
that model. (As in the above B and C sections of this problem, although the strings are different, the symbol ordering list will reorder itself to adapt the strings to generate the probability skew.)

(E) Yes.
Note that in the English language, certain pairs will appear very frequently, (such as “an”, “th”). Based on this observation, we can use one symbol to represent such a symbol pair. When doing the coding, only one symbol is used to represent such two symbols, where originally it needs to code the symbols twice. By using an intelligent way to selecting such symbol pairs, the saving will surely out-weight the overhead caused by the codebook.

**Problem 2:**

(A) Let’s write the encrypted message of the plaintext $x$ as:

$$w = xSGP + e$$

(2.1)

To decrypt $w$, we first apply the reverse permutation of $P$ to this encrypted message. That is:

$$wP^{-1} = (xSGP + e)P^{-1} = xSG + eP^{-1}$$

(2.2)

Since the weight of $e$ is $t$, and $P^{-1}$ is just a permutation operation over $e$, thus, the weight of vector $eP^{-1}$ is still $t$. Let’s think of $eP^{-1}$ as the random error in the code. Since the weight of it is $t$, according to the definition of Goppa code, we can restore the code $xS$ from the $wP^{-1}$. Since $S$ is an invertible matrix, we can easily get the plaintext from $xS$ by doing $xSS^{-1}$.

(B) Since we know $G'$, we can factor the $SGP$ from this matrix if given unbounded time and unbounded preprocessing time.

(C) Given [1024, 524, 101], Goppa code can correct at most 50 errors. Suppose that the random generated bit $e$ has weight $t$. Since there will be up to 10 extra errors which may be caused by transmission or other reason, in order to correct the code, we need to have $t + 10 \leq 50$, i.e. $t \leq 40$.

The impact of this over the encryption algorithm is that, if you want to keep the other things unchanged, then you have to limit the weight of your manually generated error $e$ no larger than 40.

This will make this algorithm easier to attack, as shown below.

One way to attack the this cryptosystem can be described as below.
As \( w = xG + e \), and \( x \) is a \( k \)-bit vector, we can reduce this to \( w_k = xG_k + e_k \), where \( w_k = w_k', w_k'...w_k' \). This means that if one picks \( k \) bit randomly from \( w \) and generate \( w_k \), if he is lucky enough so that \( e \) has all in all the corresponding bits, then he can easily regenerate the plaintext by \( x = w_k (G_k)^{-1} \). Thus, the complexity of attacking it depending on the probability of randomly selecting \( k \) bits from \( e \) and all of these \( k \) bits are 0. Obviously, this probability can be calculated using \( p = C^k_{n-r} / C^k_n \). Substitute \( t = 50 \) and \( t = 40 \), then we can calculate the probability as \( p(50) / p(40) = C^{524}_{1024-50} / C^{524}_{1024-40} \). Thus, by making \( t = 40 \), the system is easier to be attacked.

(D)

Advantages:
1. Fast. Since there is no mod operation in the whole encryption process, the speed can be very fast.

Disadvantages:
1. In order to send 524 bit message, it has to send 1024 bits. So that it’s not that efficient (high overhead).
2. The public key is too long. Public key \( G' = SGP \) is a large \( k \times n = 524 \times 1024 \) matrix.
3. Relatively easy to be attacked.

**Problem 3:**

(A)

Obviously, the matrix \( A \) has the following feature when we dismiss that additional column. Each row is a represents a separate vertex and each column stands for a separate edge. Since each edge can only connect to two vertices, each column will have exactly one 1 and one \(-1\), with all the other elements 0.

Now suppose that the basis variables in the simplex method for this matrix \( A \) include a cycle. Suppose that edges in this cycle are \( e_{a,b}, e_{a,b},...e_{a,b} \). Note that each vertex belong to the cycle will appear exactly twice. Let’s consider the sub-matrix composed of these basis variables. For those columns that corresponding to the edges belong to the cycle but in the counter-clock direction, we multiple the whole column by \(-1\). After doing that, the addition of those columns of the edges belong to the cycle will produce a 0 vector\(^1\). This means that these columns are correlated, and the sub-matrix is thus not invertible. However in simplex method, the sub-matrix composed of basis variables must be invertible. This leads to the contradiction. This proves that the edges corresponding to a set of basis variables in the simplex method for the matrix \( A \) can not include a cycle.

\(^1\)Since after the above transformations, in these columns, every row corresponding to a vertex will have only two non-zero element, one is 1 and one is \(-1\). For the other rows, the elements in the column are all 0.
(B)
No.

1. If a path has nonzero flow, it means that the edges in this path must belong to the basis variable set (if it’s not a basis variable, it will be fixed to 0).
2. Also, two distinct paths between one source and one sink can actually form a cycle. Thus, if such kind of nonzero paths exists, it means that the edges corresponding to a set basis variables in the simplex method for the matrix $A$ include a cycle, which is already proved to be impossible in the $A$.

Hence, in any of the intermediate solutions during each step of the simplex method, there cannot be two distinct paths between a source (positive $b$) and a sink (negative $b$) with nonzero flow.

(C)
Since all the vertices have nonzero $b$, the graph has to be connected in any intermediate solutions of the simplex method. And from (A), we know that there is no cycle in any intermediate solution. Thus, one step of the simplex method corresponds to adding an edge to the graph and form and form a cycle, then a edge in the cycle is selected and deleted to break the cycle.

**Problem 4:**
Suppose that the graph is composed of $n$ vertices ($V_1, V_2,...V_n$) and $m$ edges ($E_1, E_2,...E_m$). Since there are totally $n$ vertices, the maximum number of colors that should be used will be $\leq n$. Suppose we have $n$ colors available, and there are ordered to be $c_1,c_2,...c_n$.

The integer programming formulation is formulated in the following. $v_{il}=1$ if $V_i$ is labeled $l$-th color, ie, color $c_l$. Otherwise, $v_{il}=0$. And we also write $E_i$ to be $e_{i_{d}}$ with $i_s$ and $i_d$ to be two vertices that it connects. Then the graph coloring using $k$ color can be formulated as integer programming problem as following:

For each vertex, only one color is used:
$$\sum_{l} v_{il} = 1, \text{ for } i = 1...n$$

For each edge, same color cannot be used for the two vertices that it connects:
$$v_{il} + v_{lj} \leq 1, \text{ for all } (i,l) \text{ pairs where } i = 1...m, \ l = 0...k$$

And obviously,
$$v_{il} \geq 0, \text{ for all } (i,l) \text{ pairs where } i = 1...n, \ l = 0...k$$

The cost to minimize is:
$$\text{Minimize } \sum_{k=1}^{n} (n^l \sum_{i=1}^{n} v_{ik})$$
After solving the integer linear programming, we can get the minimum color that we use to label the graph to be:

$$\text{max } k \text{ where } \sum_{i=1}^{n} v_{ik} > 0$$

The reason that I use the above minimizing function is that:

1. Obviously, the number of vertices that is labeled with color $c_k$ is non-increasing with the $k$. This also means that if $\sum_{i=1}^{n} v_{ik_1} > 0$ and $\sum_{i=1}^{n} v_{ik_2} > 0$, where $k_1 < k_2$, then for all the $k$ which satisfy $k_1 < k < k_2$, $\sum_{i=1}^{n} v_{ik} > 0$.

2. If there are two feasible solutions, one uses $k_1$ colors and the other uses $k_2$ colors, where $k_1 < k_2$. Then the first one must have smaller cost than the second one because the when using $k_1$ colors, $\sum_{k=1}^{k_1} (n^k \sum_{i=1}^{n} v_{ik}) < n^{k_1} \times n = n^{k_1+1}$, while when using $k_2$ colors, $\sum_{k=1}^{k_2} (n^k \sum_{i=1}^{n} v_{ik}) > n^{k_2}$.

Thus, the whole integer programming will try to use as fewer colors as possible. This is obviously a linear programming problem with at most a polynomial number of equations and variables (in the size of $G$).

**Problem 5:**

Here the Hidden Markov Model $M$ is defined by an alphabet denoting observation variables (output variables) $O$:$\left(A,C,G,T\right)$, a set of (hidden) states $x$:$\left(A+,C+,G+,T+,A-,C-,G-,T-\right)$, a matrix of state transition probabilities $A$ which is shown in the question sheet. And a matrix of observation probabilities $B$ as following:

<table>
<thead>
<tr>
<th></th>
<th>A+</th>
<th>C+</th>
<th>G+</th>
<th>T+</th>
<th>A-</th>
<th>C-</th>
<th>G-</th>
<th>T-</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Let path $x = x_1x_2...x_n$ be a sequence of states of this Hidden Markov Model. The problem here is then given a sequence $O = O_1O_2...O_n$ of nucleotides, we try to find out the optimal path $x^*$ such that $p(O|x^*)$ is maximized. Here I use dynamic programming to solve this problem, which is actually the one developed by Viterbi.

Let’s use $P_y(i)$ to be a sequence of the state and observations which has the highest probability of the most probable path for the output sequence $O_1O_2...O_i$ and ending in state $y$ where $y \in X$. Then obvious, we have:

$$P_y(i+1) = B_{y,O_{i+1}} \cdot \max_{y \in X} \{P_y(i) \cdot A_{y,z}\}$$
Using the above equation and the initial condition \( P_{x_0}(0) \), we can calculate all the \( P_{x_0}(n) \).

\( p(O \mid x^*) \) can thus be calculated as:

\[
p(O \mid x^*) = P_{x_0}(n)
\]

Clearly this algorithm has the complexity of \( O(n) \).

There is one numerical problem with this algorithm. Since \( P_x(i) \) are the product of a series of probabilities, it can be very small so that it cannot be efficiently expressed by the floating point of the computer. One way to fix this is to use its logarithmic value as during the computation.

**Problem 6:**

Suppose we are comparing string \( A \) and string \( B \). For each cell of \( n \times m \) matrix, we keep three variables \( E, F \) and \( G \), with the following definition:

- \( E \) = optimal alignment of form \( A, B: \_ \)
- \( F \) = optimal alignment of form \( A: \_ , B \)
- \( G \) = optimal alignment of form \( A, B \)

\[
C_{i,j} = \max\{E_{i,j}, F_{i,j}, G_{i,j}\}
\]

Thus, we have:

\[
E_{0,j} = -\infty, F_{j,0} = -\infty, G_{0,0} = 0, G_{i,0} = i\beta, G_{0,j} = j\beta
\]

\[
E_{i,j} = \max\left\{G_{i-1,j} + \beta, E_{i-1,j} + \beta\right\},
F_{i,j} = \max\left\{G_{i,j-1} + \beta, F_{i,j-1} + \beta\right\},
G_{i,j} = \max\left\{G_{i-1,j-1} + c(a_i, b_j),
E_{i-1,j-1} + c(a_i, b_j) + \alpha, F_{i-1,j-1} + c(a_i, b_j) + \alpha\right\}
\]

\[
C_{i,j} = \max\{E_{i,j}, F_{i,j}, G_{i,j}\}
\]

Using the above equation, we can compute the alignment in \( O(nm) \) time for sequence of length \( n \) and \( m \).

**Problem 7:**

(A)

Suppose that there are totally \( n \) documents. We first build a document-document matrix \( A \) whose element is defined by:

\[
A_{ij} = 1, \text{ if document } i \text{ point to document } j.
\]

\[
A_{ij} = 0, \text{ otherwise.}
\]

Then we can do the calculate the truncated SVD of matrix \( A \). Depends on the rows/columns that specified (which is related to the query accuracy), we can write \( A_k \) in as:

\[
A_k = U_k \Sigma_k V_k^T
\]

\( A_k \) is the closet matrix of rank (dimension) \( k \) to \( A \).
Thus the target of the preprocessing is to build the matrix $A$ and then calculate $U_k, \Sigma_k, V_k$ and $A_k$ respectively.

(B) Let $q$ be the vector representation of a query. If there are totally $n$ documents, then $q$ is an $n$-dimensional vector. Then we transform $q$ into a vector $\hat{q}$ in $k$-dimensional space, according to the equation:

$$\hat{q} = q^T U_k \Sigma_k^{-1}$$

This projected query vector $\hat{q}$ can then be compared to the document vectors, and documents get ranked by their proximity to the $\hat{q}$ (the cosine measure can be used as a measure of the nearness), and all the documents is within certain distance can be returned as the search results.

(C) To calculate the distance between the query vector and a document in $k$-dimensional space, it will introduce $O(k)$ computation complexity. Since we have to compare it with all the $n$ documents, the complexity of a query is thus $O(nk)$.

(D) Yes.

We can use the same preprocessing as used before by matrix transposition. This is because in the above document-to-document matrix $A$, documents in the row represent the documents pointing to other documents, and the documents in the column represent the document to be pointed. Now the document-to-document matrix $B$ is the matrix where the documents in the row are the documents to be pointed and the documents in the column are the documents pointing other documents. Obviously, we have:

$$B = A = (U \Sigma V^T)^T = V \Sigma U^T = V \Sigma U^T.$$

By which we can thus get $U_k, \Sigma_k, V_k$ for $B_k$.

(E) First, we build a document-to-document matrix. For the element $(i,j)$ in this matrix, we calculate it’s value in the following way:

First, based on the term book, we find that there are $n$ terms shared by document $i$ and document $j$. Then, we find that there are $m_1$ links of document $i$ pointing to document $j$ and $m_2$ links that pointing from document $j$ to document $i$. Then we assign the element $(i,j)$ to be $\lambda_1 n + \lambda_2 m_1 + \lambda_3 m_2$.

After doing the SVD on this matrix, we can perform all the query using the previous way.