Characteristics of RNA Secondary Structure

In this lecture, we covered the topics of prediction of RNA secondary structure and detection of RNA. RNA structure is determined mainly by the formation of Hydrogen bonds between complementary nucleotides; viz. A-U, G-C and G-U. They form 3, 2 and 1 Hydrogen bonds respectively. Fig1 shows the characteristic features of the RNA secondary structure.

Fig1. RNA Secondary Structure

The secondary structure determines much of the functions of the RNA, for instance, the loops are the binding sites, and the stems hold these different binding sites together.

Context Free Grammers (CFGs)

The RNA secondary structure can be well captured by Context Free Grammars (CFGs). The reason for this is that CFGs are ideal for capturing interactions between distant pairs.
of letters in a sequence, when these interactions are in a nested structure. A pair of interacting letters are either always completely contained within other pairs of interacting letters, or in parallel to other pairs of interacting letters. This is identical in nature to parentheses. A pair of corresponding parentheses are either completely contained within another pair, or in parallel to other pairs of corresponding parentheses. Such structures are very well modeled by CFGs. Such interactions cannot, for instance, be modeled using HMMs, or their syntactic counterparts, viz. Regular Expressions. Fig2 is an example of a CFG that represents the hairpin loop shown in the right of the figure.

Fig2. Example of a CFG

This is a simple CFG that only derives the sequence shown in the right of Fig2. How exactly the sequence is derived is shown in Fig3, and the red arrow shows the direction in which the derivation proceeds.

![CFG Diagram](image)
Now consider the following CFG.

\[
S \rightarrow aSu : 3 | uSa : 3 \\
gSc : 2 | cSg : 2 \\
gSu : 1 | uSg : 1 \\
SS : 0 \\
aS : 0 | cS : 0 | gS : 0 | uS : 0 | \varepsilon : 0
\]

where \( \varepsilon \) is the empty string: “”. This grammar describes RNA secondary structure. To understand this, consider production rules of the type \( a_1Sa_2 \) to represent the fact that \( a_1 \) and \( a_2 \) are paired with one another, and the rules of the type \( a_1S \) to represent those letters that aren’t paired with anything. Each rule is also given a score, which directly corresponds to how “good” the corresponding base pair formation is considered. In this case, each production rule is given a score equal to the number of Hydrogen bonds the corresponding pairing contributes to the overall structure. The more the number of Hydrogen bonds, the more stable the structure. The Nussinov algorithm, as discussed in the previous lecture, then finds the optimal parse of a string with this grammar. The Nussinov algorithm is a simple DP algorithm, and is described in Fig 4.

### The Nussinov Algorithm

**Initialization:**

\[
F(i, i-1) = 0; \quad \text{for } i = 2 \text{ to } N \\
F(i, i) = 0; \quad \text{for } i = 1 \text{ to } N
\]

\[
S \rightarrow a | c | g | u
\]

**Iteration:**

For \( i = 2 \) to \( N \):

For \( i = 1 \) to \( N - 1 \)

\[
F(i, j) = \max \{ F(i+1, j - 1) + s(x_i, x_j), F(i, k) + F(k+1, j) \} \\
\quad \max \{ i \leq k < j \}
\]

\[
F(i, i) = 0; \quad \text{for } i = 1 \text{ to } N
\]

**Termination:**

Best structure is given by \( F(1, N) \)

---

Fig 4. The Nussinov Algorithm

Here, \( F(i, j) \) is the score of the optimal fold, or configuration, of the substring \( x_i \ldots x_j \). The different productions of the CFG that the different steps of the Nussinov algorithm
correspond to are shown on the right hand side of the steps. \( F(1, N) \) gives us the score of the optimal parse of the sequence. Note that the CFG is a very simplistic model of the RNA secondary structure. It doesn’t account for things like loop size, composition of the loops, etc. But it gives a reasonable prediction of what the secondary RNA structure is.

**Stochastic CFGs (SCFGs)**

Stochastic CFGs are CFGs where each production rule has an associated probability. They are very useful in modeling the RNA secondary structure. To draw an analogy, Stochastic CFGs are to CFGs what HMMs are to finite automata. The counterparts of HMM states in a stochastic CFG are the non-terminals. The rules with a non-terminal on the left hand side are equivalent to the transitions “out of” that non-terminal. Hence, the probabilities of the rules with some non-terminal \( V \) on the left hand side should be such that they add up to 1. As an offshoot, it should become very clear that it would be extremely undesirable to have a non-terminal without any rules going “out of” it, since any derivation including that non-terminal would never terminate.

With such an analogy, a number of computational problems analogous to those for HMMs become apparent. They are

- Finding the optimal alignment between a sequence and a SCFG (Decoding)
- Finding the probability that a sequence is generated by a given SCFG (Evaluation)
- Given a set of sequences, estimate the parameters of the SCFG (Learning)

Decoding, for instance, is interesting because the most likely parse would give us the locations of the Hydrogen bonds in the optimal RNA secondary structure. We now consider each of these problems.

**Evaluation**

Recall how evaluation was done using HMMs. We defined two dynamic programming (DP) variables; \( \text{forward} \) and \( \text{backward} \). We then used these 2 variables to find the probability that the given string was produced by the HMM.

In particular,

\[
\begin{align*}
\text{Forward:} \quad & f_l(i) = P(x_1 \ldots x_i, \pi_i = l) \\
\text{Backward:} \quad & b_k(i) = P(x_{i+1} \ldots x_N \mid \pi_i = k)
\end{align*}
\]

Then, we could compute the probability of a string \( x = x_1 x_2 \ldots x_N \) being generated by the model to be

\[
P(x) = \sum_k f_k(N)a_{k0} = \sum_k a_{0k} e_k(x_1) b_k(1)
\]

For SCFGs, we define 2 analogous DP variables, viz. \text{Inside} and \text{Outside}.
The *Inside* variable \(a(i, j, V)\) gives the probability that the substring \(x_i \ldots x_j\) of \(x\) is generated by the non-terminal \(V\). The *Outside* variable \(b(i, j, V)\) gives the probability that the entire string \(x\) *except* for the substring \(x_i \ldots x_j\) is generated by \(S\) (the root non-terminal) and the excluded part is rooted at \(V\). However \(b(i, j, V)\) does not calculate the probability that \(x_i \ldots x_j\) is generated by \(V\).

We now define a normal form for CFGs called Chomsky Normal Form (CNF). This is useful because for one, it has been proved that every CFG can be converted to an equivalent one in CNF, and moreover, once we assume that CFGs are in Chomsky Normal Form, various algorithms can be described and analyzed very elegantly.

**Chomsky Normal Form (CNF)**

In CNF, the only productions allowed are those of the following type:

\[
\begin{align*}
X & \rightarrow YZ \\
X & \rightarrow a
\end{align*}
\]

That is, the only allowed productions are those where a non-terminal goes either to 2 non-terminals or to a terminal. As an example of converting a CFG to an equivalent one in CNF, consider the following CFG and an associated derivation tree for a string in the grammar, viz. \(aabbbc\).

\[
\begin{align*}
S & \rightarrow ABC \\
A & \rightarrow Aa \mid a \\
B & \rightarrow Bb \mid b \\
C & \rightarrow CAc \mid c
\end{align*}
\]

Clearly this CFG is not in CNF, since the 4 productions, viz., \(S \rightarrow ABC\), \(A \rightarrow Aa\), \(B \rightarrow Bb\) and \(C \rightarrow CAc\) do not conform to the conditions of the Chomsky Normal Form. The equivalent CFG in CNF and the derivation tree for the same string above are
Consider another example of a CFG that is not in CNF.

\[
S \rightarrow AS' \\
S' \rightarrow BC \\
A \rightarrow AA \mid a \\
B \rightarrow BB \mid b \\
C \rightarrow DC' \mid c \\
C' \rightarrow c \\
D \rightarrow CA
\]

The slight twist here is the production \( A \rightarrow C \). We can’t use the trick here of introducing new non-terminals to handle it, since the right hand side is just one non-terminal long, and empty productions are not allowed in the CNF. The way we deal with this is, first all the productions for the other non-terminals are handled. So we get,

\[
S \rightarrow AS' \\
S' \rightarrow BC \\
B \rightarrow B'B \mid b \\
B' \rightarrow b \\
C \rightarrow C'C'' \\
C' \rightarrow c \\
C'' \rightarrow CD \\
D \rightarrow d
\]

Once we are done with this, we see that \( A \) goes to either \( aA \) or to \( C \)

Hence, we get

\[
A \rightarrow A'A \\
A' \rightarrow a
\]

from the first one. Now, since \( A \rightarrow C \) and \( C \rightarrow C'C'' \mid c \), we get

\[
A \rightarrow C'C'' \mid c
\]

In this way, we have converted the original CFG into an equivalent one in Chomsky Normal Form. (Note, by equivalent we mean that every string produced by one CFG is produced by the other, and vise-versa)
Now that we know that every CFG can be converted to an equivalent one in Chomsky Normal Form, we discuss various algorithms that will be analyzed with the assumption that the CFG is in Chomsky Normal Form.

### The Inside and Outside Algorithms

Recall that the Inside variable $a(i, j, V)$ gives us the probability that $x_i\ldots x_j$ is produced by non-terminal $V$. This problem can be solved recursively as follows. If there is a production $V \rightarrow XY$, then the probability of $V$ producing $x_i\ldots x_j$ by using this rule is the probability that $X$ produces the first part of the substring, and $Y$ produces the rest. That is, there is some $k$ between $i$ and $j$, such that $X$ produces $x_i\ldots x_k$ and $Y$ produces $x_{k+1}\ldots x_j$. Hence, the probability of $V$ producing $x_i\ldots x_j$ by the production $V \rightarrow XY$ is given by summing up the product $a(i, k, X) \ a(k+1, j, Y)$ for all different values of $k$. It follows clearly that the total probability of producing $x_i\ldots x_j$ from $V$ is given by

$$a(i, j, V) = \Sigma_X \Sigma_Y \Sigma_k \ a(i, k, X) \ a(k+1, j, Y) \ P(V \rightarrow XY)$$

We could sum up all the values since they all represent the probabilities of disjoint events. Fig5 below is a good way of visualizing the above formula.

---

**Fig5. Visual Representation of the Inside variable**

Remember that we are considering only CFGs in Chomsky Normal Form. It is clear from the above figure that the formula is used when $V$ is not a leaf node. When $V$ is a leaf, then we know that $i = j$ (since in CNF, a non-terminal can go to 2 non-terminals or one terminal, and since it is a leaf, it must go to a single terminal.) In this case, $a(i, j, V)$ is just the probability associated with the transition $V \rightarrow x_i$. Formally, the steps involved in the Inside algorithm are as shown below.
The running time of this algorithm is \( O(N^3k^2) \) where \( k \) is the number of non-terminals in grammar and \( N \) is the length of the sequence. The space complexity of the algorithm is \( O(N^2k) \).

When the algorithm terminates, \( a(1, N, S) \) gives us \( P(x \mid \theta) \), i.e. the probability that the string \( x \) was generated by the grammar. This is exactly what the problem of Evaluation was defined to be.

We handle the Outside algorithm in a similar way. Recall that the Outside variable \( b(i, j, V) \) is the probability that the entire string \( x \) except for the substring \( x_i \ldots x_j \) is produced by the root non-terminal \( S \), and the “gap” is rooted at \( V \). That is,

\[
b(i, j, V) = \text{Prob}(S \text{ produces } x_1 \ldots x_{i-1} V x_{j+1} \ldots x_N)
\]

It does not, however, calculate the probability that \( x_i \ldots x_j \) is produced by \( V \). We assume that \( V \) is some general node other than \( S \). Now consider those productions where the non-terminal \( V \) is the second non-terminal on the right hand side, i.e., productions of the type \( Y \rightarrow XV \). Then, by looking at the tree diagram in Fig7. and by using arguments similar to the ones used for the Inside algorithm, it should be clear that

\[
b(i, j, V) = \sum_X \sum_Y \sum_{k<i} a(k, i-1, X) b(k, j, Y) P(Y \rightarrow XV)
\]
This is for the case for productions where $V$ is the second non-terminal on the right hand side. There is an analogous expression for the case where $V$ is the first non-terminal on the right hand side of the productions. It should be easy to see that this expression is

$$b(i, j, V) = \sum_X \sum_Y \sum_{k>j} a(j+1, k, X) \ b(i, k, Y) \ P(Y \rightarrow VX)$$

Formally, the steps involved in the Outside algorithm are shown below.

**Initialization:**

- $b(1, N, S) = 1$
- For any other $V$, $b(1, N, V) = 0$

**Iteration:**

```
For i = 1 to N-1
    For j = N down to i
        For V a nonterminal
            $b(i, j, V) = \sum_X \sum_Y \sum_{k<j} a(k, i-1, X) \ b(k, j, Y) \ P(Y \rightarrow VX) +$
            $\sum_X \sum_Y \sum_{k>j} a(j+1, k, X) \ b(i, k, Y) \ P(Y \rightarrow VX)$
```

**Termination:**

It is true for any $i$, that:

$$P(x_1 | \theta) = \sum_X b(i, i, X) \ P(X \rightarrow x_i)$$
Now that we have the Inside and Outside variables, we move onto another one of the computational problems we had defined, namely that of Learning.

Learning for SCFGs

Recall that in the learning problem, what we are trying to do is estimate the parameters of a SCFG. We solve this problem using the method of Expectation Maximization.

Let \( c(V) \) be the count, or the number of times, that the non-terminal \( V \) is used in the parse of \( x = x_1 \ldots x_N \). That is, you sum “probabilistically” the number of times \( V \) appears in all possible parse trees for the string \( x \). From the definitions the Inside and Outside variables, it should be clear to see that

\[
c(V) = \frac{1}{P(x|\theta)} \sum_{1 \leq i \leq N} \sum_{i \leq j \leq N} a(i, j, V) b(i, j, V)
\]

In a similar manner, we can also calculate the expected number of times that the production \( V \rightarrow XY \) is used in the derivation of string \( x \). This is given by

\[
c(V \rightarrow XY) = \frac{1}{P(x|\theta)} \sum_{1 \leq i \leq N} \sum_{i < j \leq N} \sum_{i \leq k < j} b(i, j, V) a(i, k, X) a(k+1, j, Y) P(V \rightarrow XY)
\]

Using these values, we can now re-estimate the probabilities of the various transitions. This is given by the following expressions.

\[
P_{\text{new}}(V \rightarrow XY) = \frac{c(V \rightarrow XY)}{c(V)}
\]

\[
P_{\text{new}}(V \rightarrow a) = \frac{c(V \rightarrow a)}{c(V)} = \frac{\left( \sum_{i : x_i = a} b(i, i, V) P(V \rightarrow a) \right)}{\left( \sum_{1 \leq i \leq N} \sum_{i < j \leq N} a(i, j, V) b(i, j, V) \right)}
\]

We now look at the 3rd computational problem we had defined, namely that of Decoding.

Decoding: The CYK Algorithm (Cocke-Younger-Kasami)

In the decoding problem, what we are trying to find is the most likely parse of a string \( x \) given the SCFG. This can also be viewed in a way as the most likely “alignment” of SCFG \( G \) to string \( x \). The intuition behind this method is, we define a DP variable \( \gamma(i, j, V) \) that gives the likelihood of the most likely parse of \( x_i \ldots x_j \) rooted at non-terminal \( V \). This is calculated in a recursive manner using the maximum likelihood of the transitions out of \( V \). It should be clear to see then, that on termination, \( \gamma(1, N, S) \) will give us the likelihood of the most likely parse of \( x \) by the grammar \( G \). If we maintain “trace-back pointers” in a manner similar to the DP algorithms for alignment, this algorithm will also give us the most likely parse.

The formal steps of the algorithm are given below, and should be self-explanatory given the intuition above.
We use the log likelihood so that we have to do summations instead of multiplications.

Fig 8. summarizes the comparison between corresponding SCFG and HMM algorithms.

<table>
<thead>
<tr>
<th>GOAL algorithm</th>
<th>HMM algorithm</th>
<th>SCFG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal parse</td>
<td>Viterbi</td>
<td>CYK</td>
</tr>
<tr>
<td>Estimation</td>
<td>Forward</td>
<td>Inside</td>
</tr>
<tr>
<td></td>
<td>Backward</td>
<td>Outside</td>
</tr>
<tr>
<td>Learning</td>
<td>EM: Fw/Bck</td>
<td>EM: Ins Outs</td>
</tr>
<tr>
<td>Memory Complexity</td>
<td>O(N K)</td>
<td>O(N² K)</td>
</tr>
<tr>
<td>Time Complexity</td>
<td>O(N K²)</td>
<td>O(N³ K³)</td>
</tr>
</tbody>
</table>

Where $K$: # of states in the HMM  
# of nonterminals in the SCFG

Fig8. Comparison between SCFG and HMM algorithms
It is clear from above that the algorithms with SCFGs are very expensive. This is the reason that SCFGs are not used other than for solving problems dealing with RNA sequences, which are usually short in length.

**Speeding up CYK**

Once again, let us look at the CFG for predicting RNA structure.

\[
S \rightarrow aS | cS | gS | uS | \epsilon \\
S \rightarrow Sa | Sc | Sg | Su \\
aSu | cSg | gSu | uSa | cSg | uSg \\
S \rightarrow SS
\]

One thing we can do now is run the learning algorithm described before to assign probabilities to the different productions, thereby getting a Stochastic CFG. The learning can be done using known RNA structures. Then we can do things like use the CYK algorithm to determine the most likely RNA folding. As mentioned before, this is a very simplistic CFG for RNA structure prediction. It does not, for instance, model loop size or loop composition at all. But if this is the SCFG that we are going to work with, we can modify the CYK algorithm to run much faster than it does in the general case. The algorithm’s structure is very similar to that of the Nussinov algorithm, and should be easy to follow. The formal steps involved in the recursive algorithm are given below.

**Initialization:**

\[
\gamma(i, i-1) = -\text{Infinity} \\
\gamma(i, i) = \log P(x_i | S)
\]

**Iteration:**

For \(i = 1\) to \(N-1\)

For \(j = i+1\) to \(N\)

\[
\gamma(i, j) = \max \left\{ \gamma(i+1, j-1) + \log P(x_i | S, x_j), \gamma(i+1, j) + \log P(S | x_i), \gamma(i, j-1) + \log P(x_i, S), \max_{i < k < j} \gamma(i, k) + \gamma(k+1, j) + \log P(S | S) \right\}
\]

The time complexity is now just \(O(N^3)\).

As mentioned, this model is a very simplistic one. The Zuker algorithm is an algorithm that is amongst the best known for prediction of folding and RNA structure. It basically uses a more elaborate model of the energy of RNA folding. It does not use a SCFG per se, but can be expressed easily as a SCFG. It has the following enhancements over the simplistic model we have seen so far.
The Zuker algorithm

- models energy between pairs of base pairs, rather than just the energy of base pairs. This lends more accuracy to the prediction. (This is just like modeling pairs of letters in HMMs as opposed to modeling just single letter emissions)
- has separate scores for bulges of different sizes
- uses separate scores for loops of different sizes and compositions
- uses separate scores for interactions between the stem and the beginning of a loop

If one thinks about it, all these are just local interactions between few letters, and can all be modeled using SCFGs without much trouble, and then trained using real data. It must be kept in mind, though, that in order to be able to accommodate these differences in our earlier simplistic model, we would need to introduce more non-terminals, and because of this we face a hit in efficiency due to the $k^3$ term in the running time. This brings us to the final topic for the lecture.

Methods for Inferring RNA fold

There are several methods for actually finding the RNA structure. One way is to use experimental methods like NMR (nuclear magnetic resonance) or Crystallography. These are expensive in general but find the RNA structure pretty accurately. Another way is the computational way. We could, for instance, use the different algorithms developed in this lecture, like the Nussinov algorithm, Zuker algorithm, SCFGs and so on. A computational method that is more commonly used is that of multiple alignment. The idea is the following. Say you are given a number of homologous sequences that are well aligned, and you know that the sequences correspond to RNA structures. Now, because of evolution, RNA sequences change over time. But one thing that shouldn’t change is the pattern of base pairs. Fig9. demonstrates this point.

![Fig9. Multiple Alignment to find RNA structure](image)

If $i$th and $j$th positions are always base paired and covary, then they are likely to be paired.
Hence, though the letters in a column in the multiple alignment may be different, if two columns are paired in one sequence, they should also be paired in the others. Evolution basically ensures that when a mutation occurs in a position that is paired, a complementary mutation also occurs in the position it is paired with. Such patterns are very good indications that the corresponding positions are paired in the RNA structure. A way to do this quantitatively is to measure the mutual information between the pairs of columns in the multiple alignment. It is a measure of entropy, and is defined as follows.

\[
M_{ij} = \sum_{a,b \in \{A,C,G,U\}} f_{ab}(i,j) \log_2 \frac{f_{ab}(i,j)}{f_a(i) f_b(j)}
\]

where \(f_{ab}(i,j)\) is the number of times the pair \((a, b)\) are in position \((i, j)\). So now, given a multiple alignment, we can infer a secondary structure, that is, a pattern of base pairs that maximizes the total mutual information among the columns of the alignment. We could restrict the pairs \((a, b)\) to only the complementary base pairs, but the above expression works fine as well. In practice, this is done using a manual Expectation Maximization (EM) technique. We get a multiple alignment, find the co-variances as discussed above, then deduce the structure to some extent, manually improve the alignment based on the co-variance information we have, find co-variances again, and continue in this fashion till we converge to a good structure.