PREDICTIONS USING HIDDEN MARKOV MODELS AND STOCHASTIC CONTEXT-FREE GRAMMARS

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By
Courtney Magnuson

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Mathematics

July 2024

Fargo, North Dakota

NORTH DAKOTA STATE UNIVERSITY

Graduate School

Title

PREDICTIONS USING HIDDEN MARKOV MODELS AND STOCHASTIC CONTEXT-FREE GRAMMARS

By

Courtney Magnuson

The supervisory committee certifies that this paper complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Torin Greenwood

Chair

Dr. Jessica Striker

Dr. Janet Page

Dr. Katherine Duggan

Approved:

July 6, 2024

Date

Dr. Friedrich Littmann

Department Chair

ABSTRACT

In systems where certain qualities are not directly observable, hidden Markov models provide a way to represent both the observable and hidden data. For systems behaving as Markov chains, the observable data can be analyzed to make predictions about the underlying system. The prediction process utilizes HMMs and a dynamic programming method called the Viterbi algorithm. The predictions are also run against simulation data to determine the accuracy of the algorithm. The accuracy measurements used here are positive predictive value and sensitivity, both of which tell about the likelihood of a prediction being correct. Similarly, for RNA strands, their three-dimensional structure is modelled using a two-dimensional approximation called a secondary structure. This secondary structure is predicted by used stochastic context-free grammars, a generalization of HMMs. The context-free grammars act in place of Markov chains, and the CYK algorithm acts analogously to the Viterbi algorithm in making predictions using SCFGs.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

As described in [3], hidden Markov models (HMMs) have a wide variety of applications. HMMs are a foundation for making probabilistic models of linear sequences, and are often utilized in areas such as DNA sequence analysis, speech [5], [4] and handwriting pattern recognition [1], economics [7], and statistical mechanics. HMMs rely on local dependence within the linear sequence, where consecutive events are related to one another. In the situation that these events are independent, HMMs are useless.

In handwriting recognition, a reader attempts to decode the message by looking at the written symbols and identifying them as letters or punctuation marks. The written symbols are the emissions, and the intended letters are the hidden states. HMMs provide a way to represent both the hidden and emitted qualities when dependency is local; in this case, the written symbols are structured within words, and thus are only dependent on the few symbols surrounding them. In economics, HMMs are used for analyzing the stock market to help traders identifying when prices will be low (and thus, a good time to buy) or high (and thus, a good time to sell), by modeling possible factors responsible the rising and falling of prices. In Section 2.2, we introduce a simple example of an HMM, where a casino has an unfair dice game.

As the number of sequences that can be considered in HMMs can be exponentially large, the use of dynamic programming simplifies the analysis process. Dynamic programming aims to narrow the scope of consideration, where redundant calculations are avoided (see Section 2.3), and algorithms can be run more quickly. There will also be a discussion on the accuracy of one dynamic programming method called the Viterbi algorithm, using measure of positive predictive value and sensitivity. For more on these accuracy measurements, see [9].

A specific application of HMMs is in RNA secondary structure analysis. The structure of RNA consists of paired elements. These paired elements are often located a distance apart within a sequence, making representation by HMMs directly a challenge. Instead, context-free grammars are used. In Section 3.1.1, we see that these grammars create an underlying tree structure for generated sequences, which allow pair to easily be tracked.

The accuracy of the RNA strand predictions is of importance, as mutations in RNA are common, and can have a variety of impacts on an individual's health. One such example is the mutation leading to the COVID-19 epidemic. The features within the RNA mutations affect the virulence and transmissibility of the virus. Because of the effects viruses have, an accuracy of RNA secondary structure prediction is crucial. For more information, see [10].

Finally, the use of stochastic context-free grammars (SCFGs) and the Cocke-Younger-Kasami algorithm provides one mean of predicting RNA secondary structure. While this is the method of focus here, the most common method among researchers is to use energy models. Energy models experimentally determine the amount of energy required to break bonds along segments of RNA, and compare results to the accepted minimums for different structural elements. For more information on the energy in RNA folding, see [8]. However, the appeal to SCFGs is that they are easier to handle combinatorially and have fewer parameters.

Predictions using SCFGs can be done by the CYK algorithm, which provides a two-dimensional analysis of the sequences of nucleotides, reading as strings of As, Cs, Us, and Gs. The CYK algorithm used the strings to determine the most probable structure (or parse tree) of the RNA. While the CYK algorithm is useful, it has been shown in [11] that the accuracy measures of the grammars are low, and that there is still much work to be done to improve the quality of predictions.

2. HIDDEN MARKOV MODELS

When studying a system that has underlying information, it may be useful to use hidden Markov models (HMMs) to predict what the underlying information is. An HMM represents a system that behaves like a Markov chain, but also had additional qualities. HMMs are used in handwriting recognition, where an observer or computer is attempting to decode what a hand-written message says, based on the written shapes of the letters and the most likely combinations of letters possible [1]. Markov chains and HMMs are also used to analyze financial markets, where traders attempt to buy and sell stocks when prices are low and high, respectively [7]. In Section 2.2.1, we look at a model of a casino with an unfair dice game.

2.1. Markov chains

A Markov chain is a stochastic model describing a sequence of events within a system. The system is closed, where there is a finite number of states that can be obtained, and the system transitions between states in a discrete manner. These observed states are denoted π_i, and the transitions from state to state create a sequence of events. This sequence of events is called a path, denoted $\pi = \pi_1\pi_2\pi_3\cdots\pi_l\pi_k\cdots$, where π_k is the state of the system at the k^{th} observation. At each state i, there is a probability a_{ij} that at the next observation, the system will have moved from i to state j. The chain is thus characterized by the parameters $a_{ij} = P(\pi_l = i|\pi_k = j)$. These transition probabilities can be represented in a transition matrix $\mathbf{B}$, where the entry $b_{i,j}$ is the probability a_{ij}. The notation describing Markov chains and Markov models follows the presentation in [2].

Example 2.1.1 *Let's say that a beaver lives in a simple habitat that consists of a pond P, a swamp S, and a forest F. The beaver's whereabouts are observed and recorded hourly. After studying the beaver's activity, it was found that sequence can be modeled by a Markov process; the beaver's current location (or current state) relies only on its previous one. After an observation where the beaver is spotted in the swamp, there is a 40% chance it will move to the forest, a 50% chance it will move to the pond, and a 10% chance it will remain in the swamp. These transitions, and the remaining probabilities, are shown in Figure 2.1.*

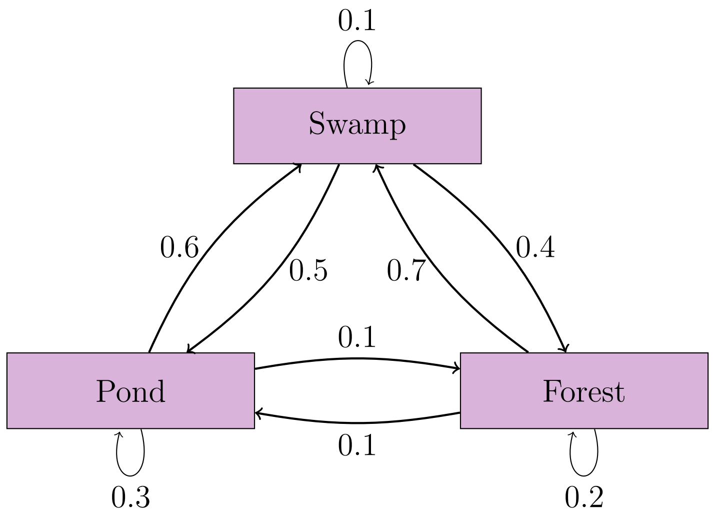

Figure 2.1. A beaver in its habitat, showing the probability of it moving between each location.

Letting the pond, swamp, and forest be states $1, 2,$ *and* 3*, respectively, the associated transition matrix is*

$$\mathbf{B} = \begin{bmatrix} 0.3 & 0.6 & 0.1 \\ 0.5 & 0.1 & 0.4 \\ 0.1 & 0.7 & 0.2 \end{bmatrix}.$$

Now, say we observe the beaver over the course of four hours (including three transitions), observing the path $SFFS$. We can determine the probability of this sequence by multiplying the three respective transition probabilities $a_{SF} \cdot a_{FF} \cdot a_{FS} = 0.4 \cdot 0.2 \cdot 0.7 = 0.056 = 5.6\%$. *The same multiplication can be used to determine the probability of any path occurring.*

Now, instead of observing the beaver at every hour, we only observe it four hours later; we observe S at observation 1 and S again at observation 4. We do not know the beaver's intermediate whereabouts. Given this information, we can try to answer a couple of questions: (1) What is the probability of this observation? (2) What are the most likely intermediate states?

We use a similar multiplication process as previously to determine the observation's probability. One method is to sum the probabilities of all possible paths of such nature. Because the number of steps and the possible locations here is small, this is a reasonable, though inefficient, task. However, with larger systems and longer paths, this is an immense, if not impossible, task to complete by brute force. Instead, another method is to neglect the intermediate steps. In this case, we only care about the path that starts at state S, transitions three times, and ends at state S. By introducing matrix multiplication, we can accomplish the same summation without acknowledging the intermediate steps. In multiplying the transition matrix n times, each entry $b_{i,j}$ in the resulting matrix represents the total probability that a path started in state i, transitioned n times, and ended in state j. The process of matrix multiplication takes the sum of all possible intermediate steps leading to the start and end states. Thus, to answer our first question, the probability of observing S three transitions *after another S is the entry $b_{2,2}$ in the matrix*

$$\mathbf{B}^3 = \begin{bmatrix} 0.304 & 0.474 & 0.222 \\ 0.384 & 0.322 & 0.294 \\ 0.288 & 0.498 & 0.214 \end{bmatrix},$$

and hence, we have a 32.2% probability.

To answer the second question, without the use of dynamic programming, we will instead consider every possible path. After considering all nine possible paths, the most likely one is $SPPS$, with a 9% chance.

It is worth noting that because the beaver's movement between locations in its habitat behaves as a Markov chain, its next location only depends on the location of current. Thus, it is of value to know its current location, as opposed to knowing no information. With knowledge of its current location, we can more accurately predict the next couple of steps it will take. Making predictions based on current information will be a key feature upcoming in Section 2.2.

As mentioned above, calculating the most probable path for the beaver was a reasonable task, as only nine calculations were required. However, as the number of possible states increase and the lengths of the paths grow longer, considering every possible option is unreasonable, if not impossible. Later in Section 2.3, we will introduce dynamic programming, which provides a solution for larger systems and for longer paths.

2.2. Hidden Markov models

In a Markov chain, the state of the system may not be directly observable. Instead, there exists a set of symbols C, and the observed events are a sequence of emitted symbols

$x = x_1 x_2 x_3 \cdots$, where each x_i is in C. Each symbol $c \in C$ has an associated emission probability, $e_i(c)$ within each state i:

$$e_i(c) = P(x_k = c | \pi_k = i). \tag{2.1}$$

The emitted symbols depend on the state of the system while the states transition as a Markov chain, as depicted in Figure 2.2.

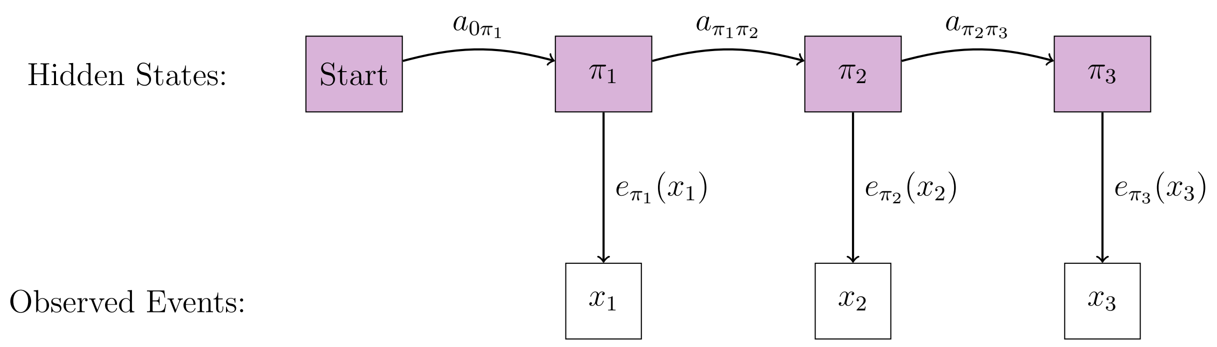

Figure 2.2. A representation of the emissions made by hidden states in an HMM and their associated probabilities.

Here, the sequence of observed events is $x = x_1 x_2 x_3$, while the state sequence is $\pi = \pi_1 \pi_2 \pi_3$. To calculate the probability of observing this sequence, we must multiply the probability of observing the state sequence by the respective emission probability at each step,

$$P(x|\pi) = a_{0\pi_1} \cdot e_{\pi_1}(x_1) \cdot a_{\pi_1\pi_2} \cdot e_{\pi_2}(x_2) \cdot a_{\pi_2\pi_3} \cdot e_{\pi_3}(x_3). \tag{2.2}$$

In general, the probability of observing a sequence x with state sequence $\pi = \pi_1 ... \pi_L$ is

$$P(x, \pi) = a_{0\pi_1} \prod_{i=1}^{L} e_{\pi_i}(x_i) a_{\pi_i \pi_{i+1}}, \tag{2.3}$$

where $\pi_{L+1} = 0$.

Here, state 0 is both the state and the end state of a sequence. In a system with an indeterminate initial state, state 0 is used to calculate probabilities that the first observation will be in state π_1, and hence given transition probabilities $a_{0\pi_1}$, denoting the probability that the system starts in state π_1. As an end state, state 0 denotes that the path is completed. The transition probabilities $a_{\pi_L 0}$ denote the likelihood of a path ending is state π_L. Thus, state 0 can only be exited on initiation and can only be entered on conclusion.

These HMMs provide a way to represent both the observed and hidden properties. While the states of the system are unknown, the observed data can be analyzed in a variety of ways to infer information about the underlying system.

2.2.1. A dishonest casino, part one

Presented as an example in Section 3.2 of [2], we have a dice game at a casino. The player of the game must predict the number value on the next die roll. Unbeknownst to the player, the game is unfair; the dealer has both a fair die and a loaded die, which has a 10% chance of rolling each number 1 through 5 and a 50% chance of rolling a 6. Thus, we have emission probabilities

$$e_F(1) = e_F(2) = e_F(3) = e_F(4) = e_F(5) = e_F(6) = 1/6 \tag{2.4}$$

and

$$e_L(1) = e_L(2) = e_L(3) = e_L(4) = e_L(5) = 1/10, \quad e_L(6) = 1/2, \tag{2.5}$$

where F and L represent fair and loaded dice, respectively. The die used depends only on which die was used on the previous roll. If the previous roll was the fair die, there is an 80% chance the next roll will also use the fair die. If the previous roll was with the loaded die, there is a 70% chance the next roll will also use the loaded die. Otherwise, the dealer will switch dice. The emission and transition probabilities are depicted in Figure 2.3, where the listed probabilities will be referred to as the "default" values.

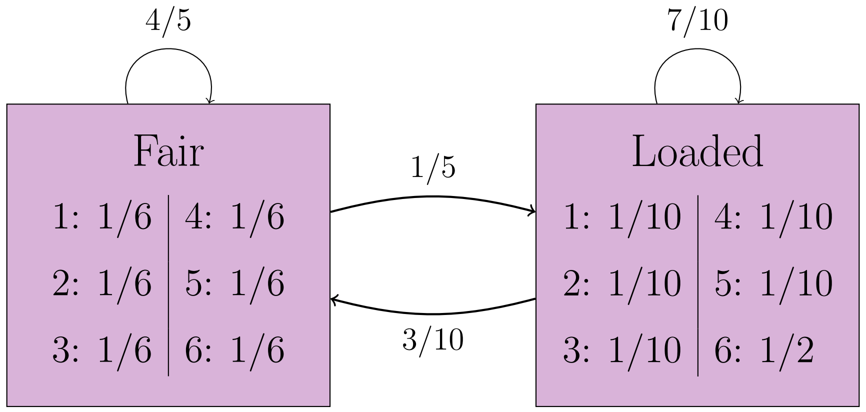

Figure 2.3. The associated transition and emission probabilities for our dishonest casino.

The corresponding transition matrix is

$$\mathbf{B} = \begin{bmatrix} 0.8 & 0.2 \\ 0.3 & 0.7 \end{bmatrix}.$$

The switch between die is a Markov process; the die used only depends on the previous one. However, the observable data is not which type of die is being used. Instead, we observe the sequence of numbers rolled. The casino does not want us to know that two dice are being used, and, in addition, which rolls used the fair die and which used the loaded one. The state sequence (the path π) in the Markov process is hidden. Thus, the dishonest casino is an HMM.

Say a player of the game sees ten rolls, observing sequence of numbers $x = 5653641236$. They do not know that the game is a dishonest one; they do not know that two dice are used, and they only see the numbers rolled. One possible (hidden) state sequence for the observed ten rolls is $\pi = FLLFFFLLL$. The sample observation, using the emission and transition probabilities from above, is represented in Figure 2.4.

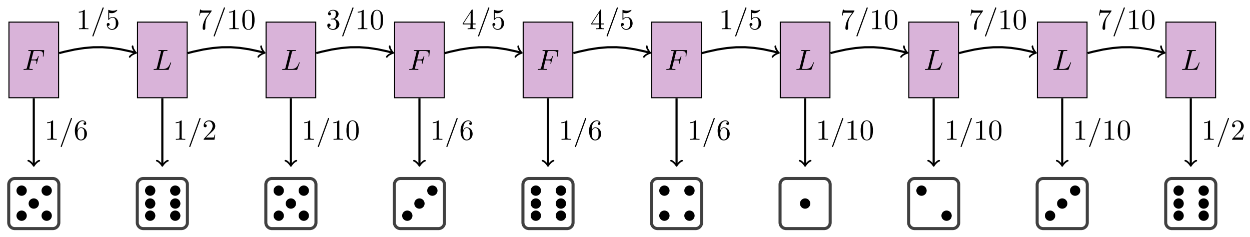

Figure 2.4. A representation of the observed die rolls, type of die used, and transition and emission probabilities associated for a sample run of 10 consecutive rolls.

You can see that the observed events depend only on the type of die being used, while the type of die used depends on which die was used previously. We will visit this dishonest casino again in Section 2.4.1, where we discuss methods of analyzing the hidden states based on the observed sequence.

2.3. Dynamic programming

Solving problems with discrete data sets often involves breaking the problem into smaller subproblems. Two methods of handling such subproblems with a computer include recursion and dynamic programming. Recursive handles the problem with a top-down approach, while dynamic programming works bottom-up, which can be seen depicted in

Section 2.3.1. With recursion, the problem is broken down by defining a function in terms of the function itself. The problem is then solved by recalling the function until the final answer is determined. Recursion leads to many repeated calculations as the function calls on itself, calculating the same value over and over again along the way, leading to a redundant process.

Dynamic programming uses a computer to store the results of the subproblems along the way. In order to avoid repeating calculations, the stored values are called until the final answer is determined. Storing intermediate values eliminates the redundancy caused by recursion.

2.3.1. Fibonacci numbers

Both recursion and dynamic programming can be used to calculate the n^{th} Fibonacci number, F_n. The Fibonacci numbers are given by the recurrence relation

$$F_n = F_{n-1} + F_{n-2} \tag{2.6}$$

with

$$F_0 = 0 \text{ and } F_1 = 1. \tag{2.7}$$

Without using dynamic programming, the recursion itself can be used to calculate F_n. This means that, in order to calculate (for example) $F_6 = F_5 + F_4$, the computer must calculate both F_4 and F_5. In order to calculate $F_4 = F_3 + F_2$ and $F_5 = F_4 + F_3$, the computer must calculate both F_2 and F_3 and F_3 and F_4, respectively. The code for the recursion follows:

where the input n must be an integer such that $n \geq 0$. The needed calculations are depicted

```
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1)+fib(n-2)
```

Figure 2.5. This code produces the Fibonacci numbers by utilizing recursion.

in Figure 2.6.

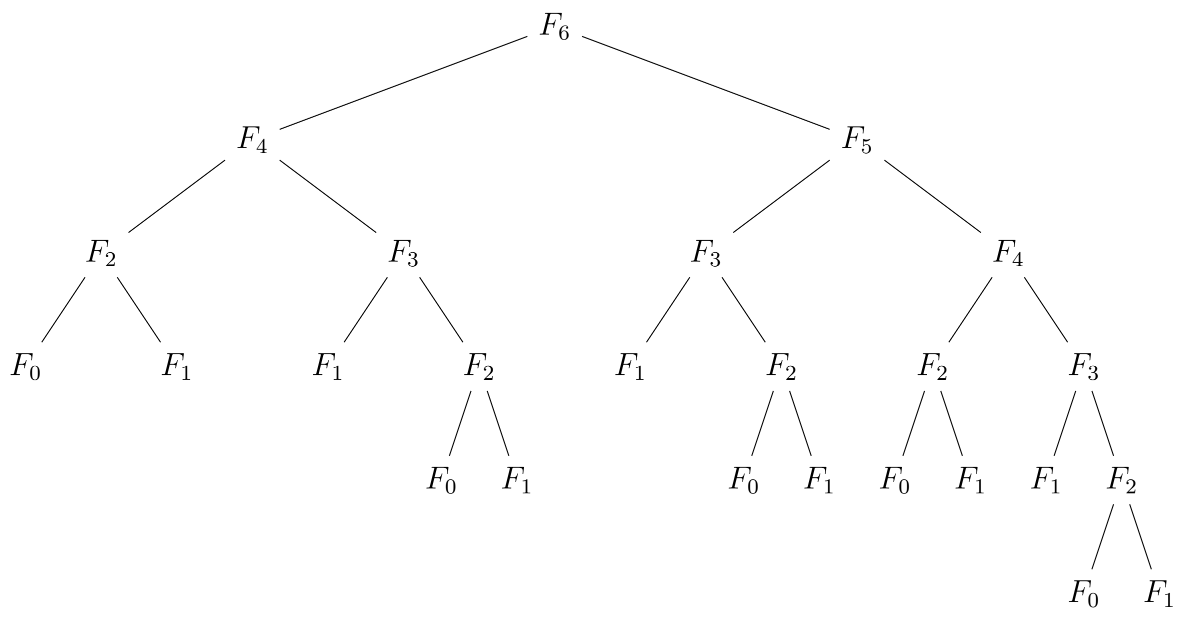

Figure 2.6. Depiction of the top-down approach of recursion used to calculate the Fibonacci number F_6.

Notice that because intermediate values are not stored in the process, the computer will make repeat calculations; every value will need to be continuously broken down until they meet the base cases of F_0 and F_1. As we calculate larger and larger values of n using recursion, the process becomes more and more redundant. (In order to calculate F_6, the single calculation of $F_2 = F_1 + F_0$ is made five times.)

Though there is no memory required from a computer to store these intermediate values, the redundancy of calculations does require the computer to remember its location in the calculation process. This amount of memory, as the number of calculations grows exponentially, is tremendous, and can cause a computer to crash quickly.

As an alternative, a computer can calculate F_n by starting from the bottom and working its way up; this time, it starts with F_0 and F_1 and calculates upwards to F_n, storing each step along the way. With this approach, no repeat calculations are required, and the computer only has to call back the previous two values stored. The code follows in Figure 2.5 and will produce a list as in Table 2.1 below. Here, each value is calculated exactly once.

```
fib = [0]*(n)
fib[0] = 0
fib[1] = 1
for i in range(2, n):
    fib[i] = fib[i-1] + fib[i-2]
```

Figure 2.7. This code produces the Fibonacci numbers by utilizing dynamic programming.

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	$\cdots$
F_n	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	$\cdots$

Table 2.1. The list of Fibonacci numbers produced from the code in Figure 2.7.

We see that recursion is a top-down approach, while dynamic programming is bottom-up. The top-down approach requires a base case to terminate the process, and will terminate after a runtime on the order of 2^n, though it does not require any storage from the computer. Unfortunately, for large enough values of n, the runtime will be too long for the base cases to be met, and thus the algorithm will not terminate in sufficient time. Recursion is often inefficient.

Dynamic programming is a bottom-up approach, with the base case being defined, and the algorithm working upwards until it terminates at the desired value of n. Because each calculation occurs only once, the runtime and storage are both on the order of n, a drastic improvement from exponential to polynomial in terms of runtime from recursion to dynamic programming.

2.4. Viterbi algorithm

When dealing with HMMs, it is often impossible to determine the state of the system by observing one singular emission. Instead, we use sequences of observations to predict the corresponding sequence of underlying states. The most common one, which we will use, is a dynamic programming algorithm called the *Viterbi algorithm* [2].

In general, when we have an HMM, there are many state sequences π that could produce any particular sequence of observed events x; for a sequence with n observations and m possible states, there are m^n possible hidden paths. However, for each observed sequence, the various possible underlying state sequences occur with different probabilities. The Viterbi algorithm narrows the scope of consideration of these paths. Working from left to right, the algorithm determines which paths are most likely to occur, and in turn eliminates the less likely ones from consideration. By increasing the length of the considered paths at each step, the Viterbi algorithm is a bottom-up approach, which determines the most probable path at the final step.

Let us call the path with the highest probability π^*, where

$$\pi^* = \underset{\pi}{\operatorname{argmax}}\; P(x, \pi), \tag{2.8}$$

which can be found recursively. Suppose we know the transition probabilities between states. For any observation beginning in state x_1 and ending in state $x_i = k$, we can find the most probable path for all states k. Let the probability of this path be denoted $v_k(i)$. Then, the probability of the next observation x_{i+1} is calculated as

$$v_l(i+1) = e_l(x_{i+1}) \max_k (v_k(i) a_{kl}), \tag{2.9}$$

where state l is the most probable state to follow state k while also yielding the observation x_{i+1}. Thus, we multiply the probability $v_k(i)$ by the transition probability a_{kl} and the emission probability $e_l(x_{i+1})$.

In application, the Viterbi algorithm uses the same logic, but works by backtracking. The full algorithm is given in Table 2.2 [2]. The index $i = 0$ provides an initial unit to

Viterbi Algorithm:	
Initialization ($i = 0$):	$v_0(0) = 1, v_k(0) = 0$ for $k > 0$.
Recursion ($i = 1...L$):	$v_l(i) = e_l(x_i)\max_k(v_k(i-1)a_{kl})$; $\text{ptr}_i(l) = \text{argmax}_k(v_k(i-1)a_{kl})$.
Termination:	$P(x, \pi^*) = \max_k(v_k(L)a_{k0})$; $\pi^*_L = \text{argmax}_k(v_k(L)a_{k0})$;
Traceback ($i = 1...L$):	$\pi^*_{i-1} = \text{ptr}_i(\pi^*_i)$.

Table 2.2. The full procedure of the Viterbi algorithm.

proceed with multiplication. The term $\text{ptr}_i(l)$ provides the means for tracking which state was the most probable state to precede observation i. That is, for the most likely path π^* up to the $i^{/nth}$ observation, $\text{ptr}_i(l)$ records state k that yielded the most probable path to state l.

It is important to note that a computer program may handle the calculations better in log space. As sequences grow larger, the repeated multiplication of probabilities yields exponentially small values, often leading to underflow errors where the computer's memory cannot sufficiently represent the value. Instead, by converting to log space, the multiplications become sums, and the key features of the algorithm are kept in tact.

2.4.1. A dishonest casino, part two

The Viterbi algorithm can find the most probable path through a sequence of die rolls. Using the model of the dishonest casino with default values described in Section 2.2.1, we use the sequence 12266. This observed sequence was generated from the (hidden) state sequence $FFLLL$ using the casino simulation, found in Appendix A.

As shown in Figure 2.8, one modification to the Viterbi algorithm is that we use v_{lk} instead of v_l. The modification, as well as providing arrows in the table, replaces the original state tracking term $\text{ptr}_i(l)$. A computer is unable to apply arrows directly, and thus the traceback terms $\text{ptr}_i(l)$ are necessary. The associated traceback table is shown below in Table 2.3.

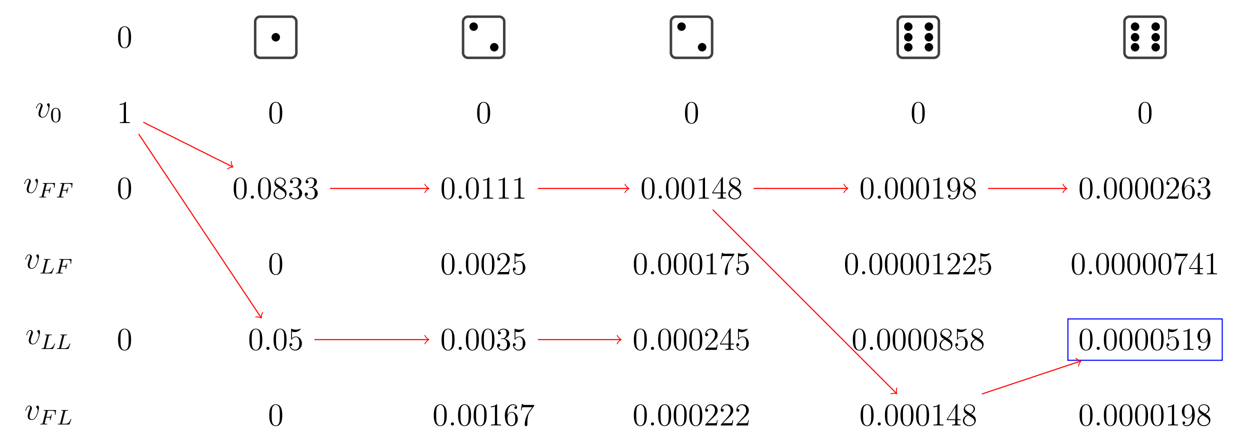
0
v_0 1 0 0 0 0 0
v_{FF} 0 0.0833 0.0111 0.00148 0.000198 0.0000263
v_{LF} 0 0.0025 0.000175 0.00001225 0.00000741
v_{LL} 0 0.05 0.0035 0.000245 0.0000858 0.0000519
v_{FL} 0 0.00167 0.000222 0.000148 0.0000198

Figure 2.8. Depiction of the Viterbi algorithm calculated probabilities, where each probability is rounded to three significant figures.

We start with the initialization step. From the initial conditions, we multiply emission and transition probabilities for the first roll showing a 2. Note that the initial transition probabilities are $a_{0F} = a_{0L} = 0.5$, meaning the casino has equal chance of starting with a fair or a loaded die. Though listed in the v_{FF} and v_{LL} rows, the probabilities listed here are starting from state zero.

Then, for each remaining transition, we have two possibilities: the next roll is a fair die, or the next roll is a loaded die. Within each of those possibilities, we have two more: the previous roll was a fair die, or the previous roll was a loaded die. For each emission, we have to compare possibilities: Is it more likely that a fair roll was preceded by a fair or a loaded roll? To answer, we determine the greater value of v_{FF} and v_{LF}, while noting (using red arrows) which of the previous probabilities was used in calculations. The same is done for loaded dice. For example, in Figure 2.8, we have $v_{FF} = 0.0111$ and $v_{LF} = 0.0025$ listed under the observation $x_2 = 2$. These are both the probabilities that the roll was with a fair die. Because $v_{FF} > v_{LF}$, it is more likely that the previous roll was also a fair die. Now, we have a red arrow pointing from $v_{FF} = 0.00148$ to $v_{FL} = 0.000148$ in the transition from 2 to 6. This means that $v_{LL} < v_{FL}$ when comparing for a loaded die in the observation $x_4 = 6$. Thus, it is more likely that a loaded die was preceded by a fair die. This process is repeated for the remainder of the observed sequence.

Once all probabilities have been calculated, the Viterbi algorithm then looks at the final column to find the overall most probable state. The traceback step can now commence, where we follow the arrows in reverse order. Here, the last roll was most likely with a loaded die, meaning the Viterbi algorithm predicts $\pi_5 = L$. Working backwards, it is most likely that the loaded die was preceded by two more loaded die rolls, which were preceded by two fair die rolls. The Viterbi algorithm thus predicts that the hidden state sequence is $FFFLL$. Here, the Viterbi prediction is not entirely correct, which may frequently be the case.

	⚀	⚁	⚁	⚅	⚅
Fair dice	0	F	F	F	F
Loaded dice	0	L	L	F	L

Table 2.3. The Viterbi algorithm traceback table associated to Figure 2.6.

To aid in application of the Viterbi algorithm, a computer program was implemented. The code first consists of defining the transition and emissions matrices. Then, a random number generator applies the transition and emission probabilities to simulate a sequence of rolls, creating both strings of the observed rolls and the hidden dice used. After generating the rolls, empty lists were created to mirror Figure 2.8. Then, the iteration process along the sequence of rolls begins, calculating the probability of each outcome, and appending results to both the probability and traceback lists. Finally, after iterating over all rolls, the traceback was implemented, creating a string in reverse order to complete the Viterbi algorithm. This code then had further features added, which allowed us to determine the accuracy of the algorithm in Section 2.5.1.

Using this code, a sample of 200 consecutive random rolls were generated. As shown in Figure 2.9, each roll used either a fair (F) or loaded (L) die, listed below the observed number. The programmed Viterbi algorithm predicted which die was used (and, hence, the state sequence) fairly well. A further analysis of the algorithm's predictive power will be provided in Section 2.5.

```
Rolls    5644536345126415652162231216436324331366633614653635151653313514625465
Dice     FFFFFFFFFFFLLLFFFFFFLLFFLFFFFFFFFFFFFFFLLLFFFFFLFFLLFFFFLFFFFFFFFFFFFLF
Viterbi  FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFLLLFFFFFFFFFFFFFFFFFFFFFFFFFFFF

Rolls    1361361543643414244612256424665433226634334555255536365445464254626166
Dice     FFFFFFFFLFFFFFFFFFFFFFFLLFFFFFFFFFFLLLLFFFFFFFLLFFLLFFFFFFFFFFFFFFFLL
Viterbi  FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFLLLLLL

Rolls    652351636663146266442563564116126665565563533634551213165543
Dice     LLLFFFLFLLLLFFLLLLFFFFLLFFFFFFFLLLLFFFFFFFFFFFFFFFFFFLFFFFFF
Viterbi  LFFFFFLLLLLLLLLLLLFFFFFFFFFFFFFFFFLLLFFFFFFFFFFFFFFFFFFFFFFF
```

Figure 2.9. The observed events for 200 consecutive rolls at our dishonest casino. Below each roll is the actual type of die, fair (F) or loaded (L), that was used. Below that is the type of die predicted by the Viterbi algorithm.

2.5. Measuring accuracy

When making predictions with an algorithm or test, it is important to consider the adequacy of the algorithm. That is, the predictive power or predictive validity must be analyzed. In Section 2.4.1, the Viterbi algorithm made a prediction about what the underlying states (or, which type of die was used). Positive predictive value (PPV) and sensitivity are two of the most common accuracy measurements used among researchers and, following suit, will be used here. To help define PPV and sensitivity, consider an example.

Say we work in healthcare, and several of our patients have sore throats. We suspect that many cases are due to strep throat. As a practitioner, we can assess our patients in two ways: through diagnostic tests and through screening tests. A diagnostic test (in this case, a strep test) provides definitive information about the presence or absence of a condition (strep throat). Though they provide definitive information, diagnostic tests may be expensive, time-consuming, invasive or discomforting to the patient. Instead, a screening test places fewer demands on the healthcare system, along with being cheaper, faster, and easier for the patient [9]. A screening test for strep throat would consist of checking for symptoms such as a sore throat, a fever, and swollen lymph nodes in the neck. Screening tests are also known for being imperfect and ambiguous; different practitioners may interpret said symptoms in different ways for different patients. Therefore, it is important to determine how well these screening tests predict the condition so the patient can proceed appropriately.

The two measurements of accuracy we will utilize are sensitivity and positive predictive value (PPV). In both cases, we want to know how well a positive screening test predicted the patient to be positive for the condition. If a patient has the condition and also received a positive screening (the patient *appeared* to have the condition), the prediction was a true positive result. If the patient has the condition but received a negative screening (the patient *did not appear* to have the condition), the prediction was a false negative. The remaining outcomes are listed in the table below, where a, b, c, and d are the number of patients receiving each outcome [9].

	Has the condition	Does not have the condition	
Predicted positive	(a) True positive	(b) False positive	← Entries for PPV
Predicted negative	(c) False negative	(d) True negative	
	↑ Entries for sensitivity		

Table 2.4. The values used for calculation of PPV and sensitivity from screening and diagnostic tests.

As shown in the table, sensitivity considers all who have the condition, while PPV considers all positive predictions. The calculations for sensitivity and PPV are

$$\text{Sensitivity} = a/(a+c) \tag{2.10}$$

$$\text{Positive predictive value (PPV)} = a/(a+b), \tag{2.11}$$

with a, b, c, and d defined as in the table. The sensitivity of a test considers the proportion of those with the condition who received a positive screening. Sensitivity disregards those who do not have the condition, regardless of their screening test, and gives the probability that someone with the condition is correctly identified in the prediction. In our healthcare example, sensitivity is the value of the correct positive screenings out of the total of patients who have strep throat. If the sample has a low sensitivity, many patients who had strep were missed in the screening test. On the other hand, a high sensitivity means few patients were with strep were missed.

The PPV of a test, by contrast, considers the proportion of the total of positive predictions who do indeed have the condition. PPV disregards all negative predictions and gives the probability that a positive prediction is correct. In our healthcare example, the PPV is the value of correct positive screenings out of the total of positive screenings. If the sample has a high PPV, most of the positive screening predictions were correct. On the other hand, a low PPV means many positive screenings were incorrect.

Example 2.5.1 *Healthcare and strep throat.*

On a given day, one practitioner saw 40 patients with sore throats. The practitioner administered both a diagnostic and a screening test on all 40 patients. The results are shown in Table 2.5.

	Has strep throat	Does not have strep throat
Screened positive	9	16
Screened negative	1	14

Table 2.5. An example of one practitioner's screening and diagnostic test results.

The corresponding accuracy measurements are then

$$\textit{Sensitivity} = 9/(9+1) = 9/10 = 0.90 \tag{2.12}$$

$$\textit{Positive predictive value (PPV)} = 9/(9+16) = 9/25 = 0.36. \tag{2.13}$$

Here, the PPV is low. The practitioner screened a high number (25) of the 40 patients as positive. Of the 25 screened positive, only 9 of them were indeed positive. This means many patients were incorrectly identified as having strep throat. On the other hand, the sensitivity is high. The practitioner was accurate in identifying 9 of the 10 patients with strep throat. One observation from this sample is that sensitivity is likely to be high when there is a large number of positive predictions.

Sensitivity and PPV both provide useful, though slightly different, information about the effectiveness of the algorithm or test being used to make predictions. It is important to use multiple measures when evaluating the effectiveness of a test; if one measurement is heavily favored, the test can be skewed in its favor. If a high sensitivity value is prioritized, then we are prioritizing having no positive conditions missed. In order to ensure few are missed, a test can be created to predict a majority (if not *all*) of the observations to be positive, as well. Prioritizing PPV can be done similarly; because PPV only considered those that are predicted positive, a test can be created to predict very few of such. Instead, it only predicts positive when it can be absolutely certain. Because of the potential for manipulation, we will use both accuracy measurements in Section 2.5.1.

2.5.1. A dishonest casino, part three

Using the same simulation of 200 rolls with default values in Figure 2.9, we can analyze the Viterbi algorithm's PPV and sensitivity. From the sample, we had the following table values:

	Die is loaded	Die is fair
Predicted loaded	18	7
Predicted fair	28	147

Table 2.6. The results for a sample of fair and loaded rolls compared to the Viterbi algorithm's predictions.

which give a sensitivity value of 0.3913 and a PPV value of 0.72. By scanning and comparing across the sequence of rolls, we can see how this occurs; more often than not, the loaded die rolls were missed in the algorithm's prediction, yielding a low sensitivity, while most of the algorithm's loaded predictions were indeed correct, yielding a high PPV.

To get a more thorough representation for measuring accuracy, we want to consider a larger sample size. The simulation of 200 rolls was repeated over a trial of 1000 runs, tallying sensitivity and PPV values throughout. Over the 1000 runs, PPV and sensitivity values where distributed into intervals of 0.05 length, and tracked for how frequently the interval values occurred, as shown in Figure 2.10. We can see that in general, the default

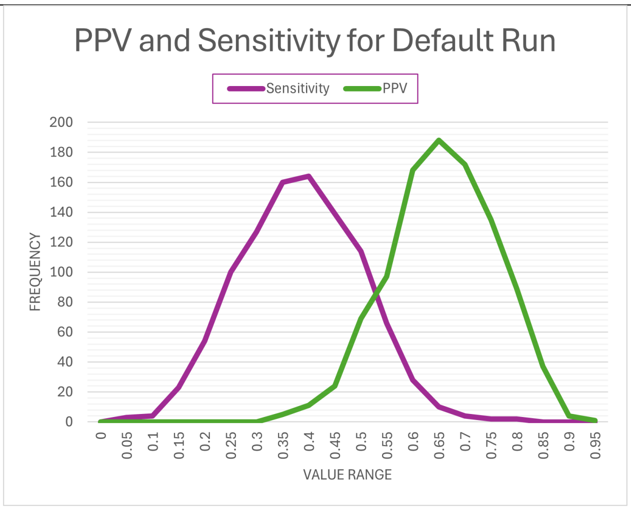

Figure 2.10. PPV and sensitivity for the default values using 1000 runs with 200 rolls each.

values lend to higher PPV values than sensitivity values, with average values being roughly 0.75 and 0.45 respectively. This is consistent with the single sample previously considered. A further analysis in Section 2.6 expands beyond the default values to test the accuracy of the algorithm given different parameters.

2.6. Further accuracy analysis

The accuracy of the Viterbi algorithm for our dishonest casino depends on the probabilities used. In our example, we used the transition matrix

$$\mathbf{B} = \begin{bmatrix} 0.8 & 0.2 \\ 0.3 & 0.7 \end{bmatrix}$$

and an emission probability of 0.5 for a 6 on the loaded die. However, these probabilities are arbitrary. As mentioned in Section 2.5.1, these values will constitute our "default" run. The following is an analysis of the accuracy of the algorithm outside of the default values. It is worth noting that in each further run, the only deviation from the default run is the value mentioned; all others were held at the default value.

The first modification from the default run was to vary the transition probability from the loaded die, changing the value of a_{LL}. The value of focus in our default run is 0.7; the probability that after a roll with a loaded die, the following roll will also use a loaded die. In further runs, probabilities of 0.9, 0.8, 0.6, 0.5, and 0.4 were used instead. Below shows the PPV and sensitivity values for each probability.

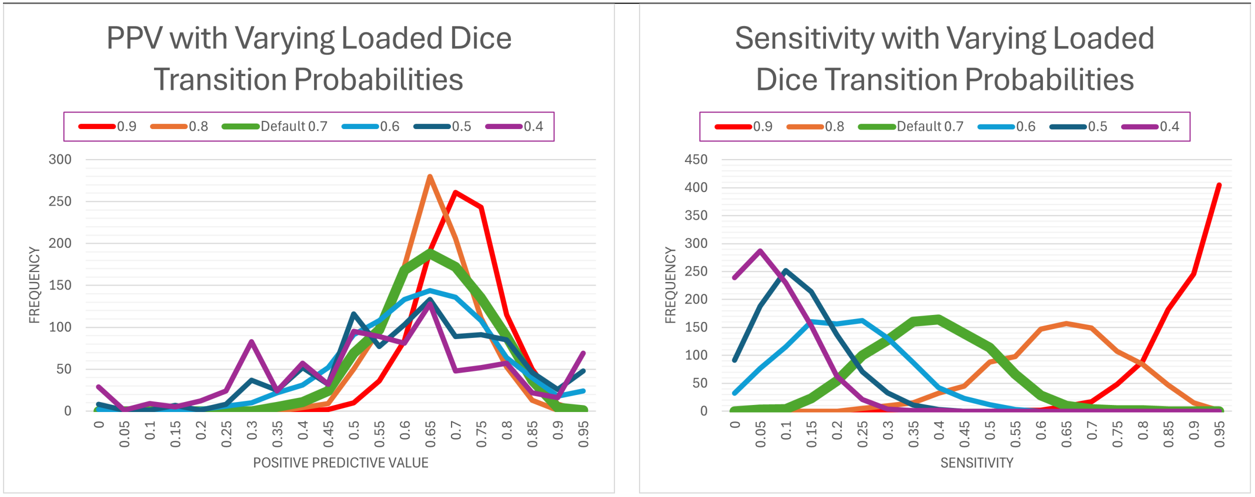

Figure 2.11. Shown are the frequency of PPV and sensitivity values over 1000 tests, each with 200 die rolls. Here, the legend values are a_{LL}, the probability that a loaded die roll follows itself.

Relatively speaking, the PPV values stayed within the same range. As the transition probability a_{LL} increased, so did the frequency of higher PPVs. As the value of a_{LL} decreased, the PPV was less consistent. This means that, in general, the algorithm was mostly consistent with the proportion of correct predictions made.

On the right side of Figure 2.11, we see the sensitivity values were less agreeable. As the a_{LL} value decreased, so did the sensitivity values. This corresponds to the fact that with lower a_{LL} values, it is less likely that loaded die rolls follow themselves, and thus switch back to a fair die more frequently. Given the more frequent switches, the algorithm is more likely to miss loaded die rolls in its prediction. On the other hand, when a_{LL} is larger, the run is loaded for more consecutive rolls, making it more likely that the algorithm predicts rolls to be loaded.

The second modification made was to vary the transition probability from the fair die, a_{FF}. Here, we return the value a_{LL} back to its default value of 0.7. The a_{FF} value of focus in the default run was 0.8, while further runs used probabilities of 0.9, 0.7, 0.6, 0.5, and 0.4 instead. Here, we see similar, though inverse, results as with the varying loaded

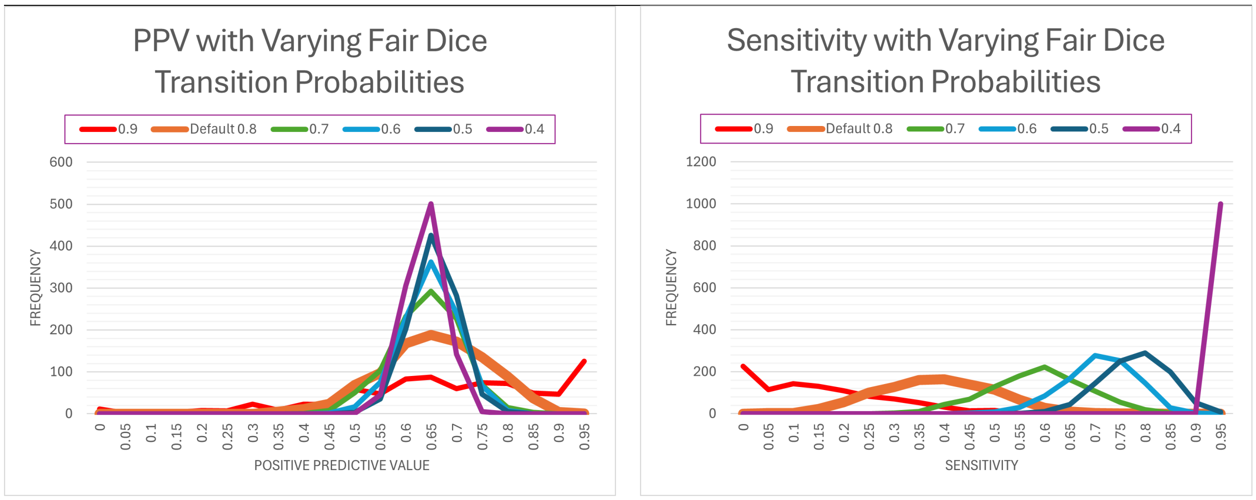

Figure 2.12. Shown are the frequency of PPV and sensitivity values over 1000 tests, each with 200 die rolls. Here, the legend values are a_{FF}, the probability that a fair die roll follows itself.

die values. The PPV values remained at a consistent average, though with lower standard deviation as the a_{FF} values decreased. The sensitivity values on the right side of Figure 2.12 show higher values for the decreasing a_{FF}. Both of these observations are likely due to the face that a lower a_{FF} values implies a greater a_{FL} value, meaning the die switches from a fair to a loaded die more frequently. The more frequent loaded die makes the algorithm more likely to predict that the roll was indeed loaded, resulting in higher and more frequent correct guesses.

The final modification made was to vary the emission probability of rolling a 6 on the loaded die, $e_L(6)$, while returning all transition probabilities to the default values. The default value of $e_L(6)$ was 0.5, while further runs used values of 1, 0.75, 0.6, 0.4, and 0. Note that while the value of $e_L(6)$ varied, the remaining probability was equally dispersed among emissions of 1 through 5. Here, we see that both PPV and sensitivity increased as $e_L(6)$

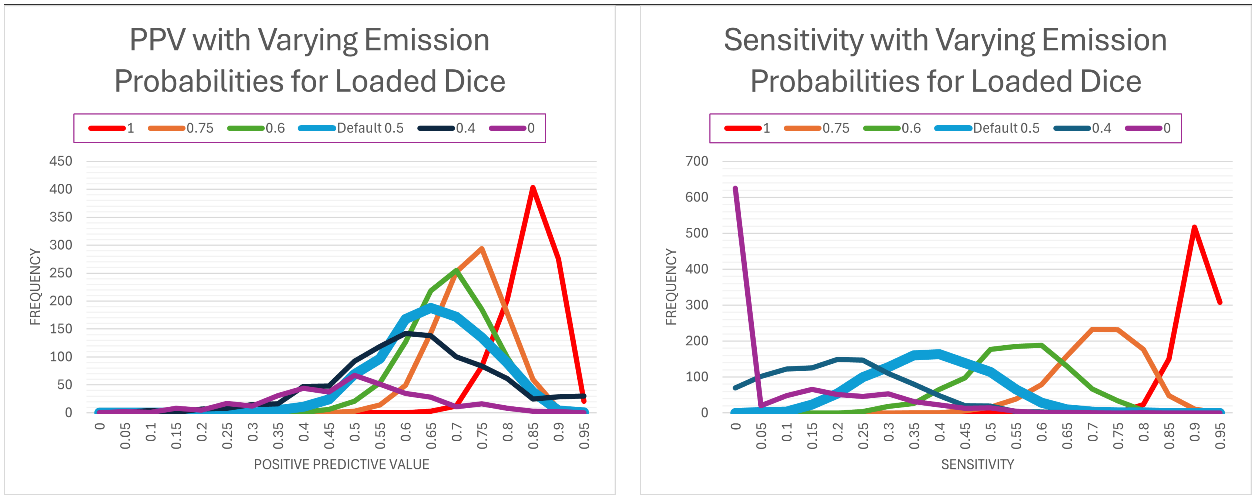

Figure 2.13. Shown are the frequency of PPV and sensitivity values over 1000 tests, each with 200 die rolls. Here, the legend values are $e_L(6)$, the probability that the loaded die rolled a 6.

increased, while PPV still remained higher in general. As $e_L(6)$ increased, the loaded die rolled more 6s, making it more likely for the algorithm to correctly identify which rolls where loaded.

Similarly, as $e_L(6)$ increased, the algorithm was less likely to miss actual loaded die rolls. Instead, as $e_L(6)$ decreased and became more similar to the fair die emissions, the algorithm was more likely to miss loaded die rolls, proving challenging to differentiate between the two.

As an aside: It is important to note that all variations were run for 1000 trials. As you may see, not all graphs show a total of 1000 for each value. This is due to the fact that our PPV and sensitivity formulas involve division. In our simulation, the PPV divides by the total number of predicted loaded dice, while the sensitivity divides by the total number of actual loaded die rolls. If either of these divisors is zero, we cannot calculate the associated value.

In all of our trials of 1000 runs over 200 rolls, we always had at least one loaded die roll. Thus, we could always calculate sensitivity. We ran into errors when the algorithm did not predict a single roll to be a loaded die. This occurred in several trials, some of which are not depicted in Figure 2.11, Figure 2.12, or Figure 2.13. The error occurred in three general trends:

- When a_{LL} values decreased. The first occurrence happened at a value of 0.6, and occurred more frequently as we decreased by 0.1. At values of 0.2 and 0.1, all 1000 runs predicted no loaded dice. Below is one partial output for $a_{LL} = 0.5$, where no loaded dice were predicted:

```
Rolls    3166244361241632634136211266235252643543616124321336653333326563416
Dice     FFFFFFFFLFFFFFFFFLFFFFFFFLLFFFFFFFLFFFFFFFLFFFLFFFFFFFFLFFFFLFFFFFF
Viterbi  FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

Figure 2.14. A sample using $a_{LL} = 0.5$ resulted in no loaded dice being predicted by the Viterbi algorithm.

- When a_{FF} values increased. The first occurrence happened at a value of 0.85, and occurred more frequently as we increased to 0.9 and 0.95, at which 665 of the 1000 runs predicted no loaded dice. Below is one partial output for $a_{FF} = 0.9$, where no loaded dice were predicted:

```
Rolls    5111326131125211566132146454641643133351651544325142135643461153266
Dice     FFFFFFFFFFFFFFFFFFFFFLFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFL
Viterbi  FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

Figure 2.15. A sample using $a_{FF} = 0.9$ resulted in no loaded dice being predicted by the Viterbi algorithm.

- When $e_L(6)$ values decreased. The first occurrence happened at a value of 0.4, and occurred again at 0.25 and 0, with, respectively, 996 and 622 of the 1000 runs predicting no loaded dice. Below is one output for $e_L(6) = 0$, where no loaded dice were predicted:

```
Rolls    6315646446625525445552632111635512114536164462523416561113545634541
Dice     LLLFFFFFFFLFFFLFFFFFFFFLFFFFLLLFFFFFFFFFFFFFFFFFFFFLLLLFFFFFFLLFFFF
Viterbi  FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

Figure 2.16. A sample using $e_L(6) = 0$ resulted in no loaded dice being predicted by the Viterbi algorithm.

The general conclusion is that because PPV remained fairly consistent throughout, positive predictions are likely to be correct. However, the algorithm is less likely to predict loaded die rolls when (1) the actual rolls are fair a vast majority of the time and (2) when the loaded die emission probabilities do not differ from the fair die by a large value, both of which have logical reasoning to justify.

3. RNA FOLDING

In order to predict the underlying structure of RNA, researchers are able to read the nucleotide sequence that runs along strands. These nucleotides often pair in specific ways, which lends to paired nucleotides being listed over a long range when the strand is read. This long-range dependency then makes it challenging to be represented by HMMs.

Instead, context-free grammars written in a type of "normal form" create a way for the pairs to be tracked. The tree structure that emerges from these grammars, such as the one in Figure 3.1, allows researchers to easily identify where the pairs originated. The "normal form" used allows for simpler calculations and representation of the combinatorial objects created, as production rules have at most two non-terminals replacing one.

3.1. Context-free grammars

A context-free grammar G (abbreviated to "grammar") is a formal grammar which is used to generate all possible strings in a given formal language. Each grammar is a four-tuple $(\mathcal{N}, \mathcal{T}, \mathcal{P}, S)$, consisting of:

- a finite set $\mathcal{N}$ of non-terminal variables,
- a finite set $\mathcal{T}$ of terminal variables that is disjoint from $\mathcal{N}$
- a finite set $\mathcal{P}$ of production rules, and
- a distinguished symbol $S \in \mathcal{N}$ that is the start symbol.

Each production rule replaces one non-terminal variable with a string of terminals and non-terminals. The notation and discussion on grammars is taken from [11].

In order to develop algorithms to analyze grammars, it is convenient to restrict the grammars to a normal form, with the most common being Chomsky Normal Form (CNF).

In CNF, the production rules allowed must be of the following form:

$$\begin{aligned} &A \to BC \\ &A \to a \\ &S \to \epsilon \end{aligned} \tag{3.1}$$

where A, B, and C are non-terminals (such that $A, B, C \in \mathcal{N}$), a is a terminal variable (such that $a \in \mathcal{T}$) and ϵ is the empty string.

Here, we will use a slight variation of CNF when defining certain grammars. Say we have the *structure generating grammar* $(\mathcal{N}, \mathcal{T}, \mathcal{P}, S)$, where $\mathcal{N} = \{T, U, V\}$, $\mathcal{T} = \{(,), .\}$, and S is the non-terminal variable T. As a modification from CNF, here production rules with the terminals "(" and ")" must be emitted simultaneously, generating closed pairs. This double emission normal form provides a more straightforward representation of the structures we will later see in Section 3.3. Now, we allow the following forms:

$$\begin{aligned} &T \to UV \\ &T \to . \\ &T \to (U) \end{aligned} \tag{3.2}$$

for any combination of non-terminal variables in $\mathcal{N}$.

Each non-terminal can have several associated production rules. For compactness, an example of non-terminals V_1, V_2, and V_3 each with multiple associated production rules will be written as:

$$\begin{aligned}
&V_1 \to V_2V_3 \mid (V_2) \mid . \\
&V_2 \to (V_1) \mid V_2V_3 \mid V_3V_2 \qquad (3.3)\\
&V_3 \to V_3V_3 \mid .
\end{aligned}$$

where the multiple rules are separated by the vertical bar "|" for each non-terminal. We will denote the set of associated production rules P_{V_1} for each non-terminal V_i. It is important to note that in grammars, the order of the variables **does** matter; the production rules $V_2 \to V_2V_3$ and $V_2 \to V_3V_2$ are not equivalent, and must be separately defined.

In the double emission normal form, the production rules will be of similar form as most recently given. The rules used are designed to avoid cyclical productions (that is, avoid combinations of rules that allow a string to regenerate itself). For this reason, the rules $T \to \epsilon$ and $T \to U$ will mostly be excluded, as they may produce countably infinite numbers of derivations of the same strings. It is worth noting that while we mostly exclude them here, strong grammars can be created with these rules present.

3.1.1. A context-free grammar

Strings (and, later, structures) are generated from the production rules, continuing until all non-terminals have turned to terminals. Say we have the grammar defined by

$$\begin{aligned}
&S \to AB \mid (B) \mid . \\
&A \to SB \mid AA \mid . \qquad (3.4)\\
&B \to (A) \mid .
\end{aligned}$$

which has the set of non-terminals $\{S, A, B\}$ and terminals $\{(,), .\}$. The following shows the steps to achieving one string in the language of the grammar:

$$\begin{aligned} \underline{S} &\Longrightarrow (\underline{B}) \Longrightarrow ((\underline{A})) \Longrightarrow ((\underline{SB})) \Longrightarrow ((\underline{A}B(\underline{A}))) \Longrightarrow ((.\underline{B}(\underline{S}B))) \Longrightarrow ((..(\underline{A}A\underline{B}))) \\ &\Longrightarrow ((..(A\underline{AB}.))) \Longrightarrow ((..(\underline{A}S\underline{B}..))) \Longrightarrow ((..(\underline{AAS}(A)..))) \Longrightarrow ((..(...(\underline{A})..))) \\ &\Longrightarrow ((..(...(.)..))) \end{aligned} \tag{3.5}$$

where each underlined variable will be changed via production rules in the next step. Here, the number of production rules replacing non-terminals at each step is random; the point of interest in application in Section 3.3 is the final derivation and the derived string. Hence, the order of production is unimportant and is shown as is for ease of understanding.

Another useful way to represent a derivation is through a *parse tree* [2]. In a complete parse tree, the root node is the start symbol S and every leaf must be a terminal variable. The internal nodes are non-terminals whose children are the production rules used, in left-to-right order. As you can see in Figure 3.1, the derived string can be seen by reading counterclockwise around the tree. Starting from S, traverse down the leftmost branches until the first leaf is reached. The label of the first leaf is the first character in the string. Then, traverse inwards until a parent node is reached. From the parent node, proceed outwards again, traversing counterclockwise until another leaf is reached, giving the second character of the string. In Figure 3.1, we first traverse down S to "($''$, then back inwards to S. From S, we traverse down again to B, then outwards to another "($''$. We continue the traverse inwards and outwards until the full string is attained, concluding upon the return to the start symbol S from the right most child node, which completes a counterclockwise loop.

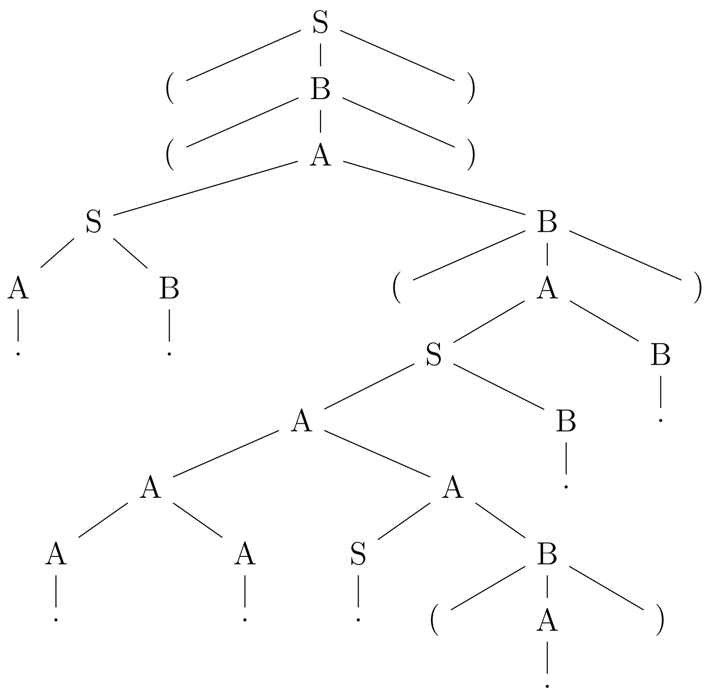

Figure 3.1. The parse tree for the string "((..(...(.)..)))", following the same production rules as given in Section 3.1.1.

The parse tree for grammars will be the representation used throughout the remainder of this paper. While these trees will be large in size, analysis using dynamic programming will utilize *subtrees*, which are fragments of the parse tree whose root is an internal node. In a subtree, all connected parts spanning from the specified root are included, as shown in Figure 3.1. The connectivity of subtrees allows dynamic programming algorithms to build larger and larger subtrees from the bottom-up.

3.1.2. Stochastic context-free grammars

In application, it is common for grammars to have probabilities associated to each production rule. For such stochastic context-free grammars (SCFGs), the probability of each derivation (string) is the product of the probabilities of each production rule used in the derivation. In other words, some production rules, and thus strings and structures, within a grammar occur more frequently than others.

Say the grammar in Section 3.1.1 has the following probabilities

$$\begin{aligned} &P[S \to AB] = 0.5 \mid P[S \to (B)] = 0.3 \mid P[S \to .] = 0.2 \\ &P[A \to SB] = 0.3 \mid P[A \to AA] = 0.1 \mid P[A \to .] = 0.6 \\ &P[B \to (A)] = 0.5 \mid P[B \to .] = 0.5 \end{aligned} \tag{3.6}$$

associated to each production rule. Then, we determine that the probability of the derivation "((..(...(.)..)))″ by multiplying the from each production rule used:

$$\begin{aligned} &P[S \to AB]^2 \cdot P[S \to (B)] \cdot P[S \to .] \cdot P[A \to SB]^3 \cdot P[A \to AA]^2 \\ &\quad \cdot P[A \to .]^4 \cdot P[B \to (A)]^3 \cdot P[B \to .]^3 = 0.6^4 \cdot 0.5^8 \cdot 0.3^4 \cdot 0.2 \cdot 0.1^2 \\ &\approx 0.000000008201. \end{aligned} \tag{3.7}$$

For this sample string of length 16, we see that the probability is very small. As mentioned in Section 2.4, using log space becomes increasingly important when implementing computer algorithms to deal with such structures as a means to avoid underflow errors.

3.2. Motzkin words

By using the production rules defined in the double emission normal form, all derived strings will consist of sets of closed parentheses supplemented by dots. Strings of such kind are called *Motzkin words*. In order for a string to be a Motzkin word, two conditions are true:

- the number of left and right parentheses must be the same, and
- when read from left to right, the running total of left parentheses is at least that of right parentheses; that is, the amount of right parentheses never exceeds the amount of left.

The implications of these conditions mean that in matched pairs, the left parenthesis always precedes the right one, and such pairs are always nested. There are several other depictions of Motzkin words, but we will use the representation shown in Figure 3.2. A nesting means

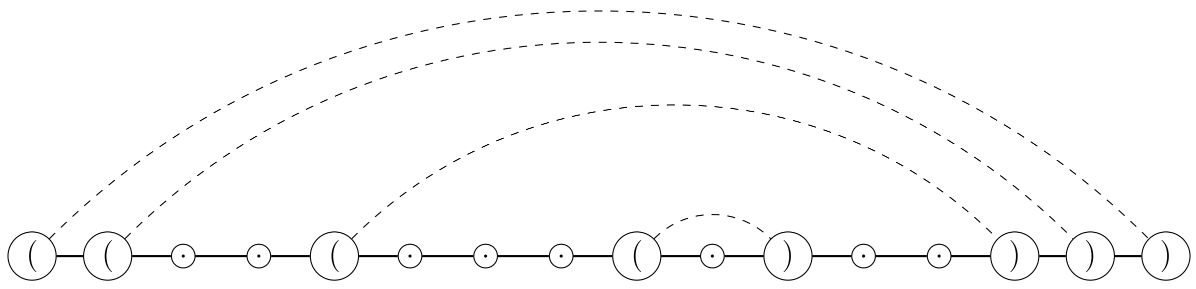

Figure 3.2. For the Motzkin word "((..(...(.)..)))″, shown is a depiction of its nested nature.

that, when connected as such, no matching pairs overlap, but not every element needs a pair. It is important to note that every subtree of a parse tree yields another Motzkin word, and not every Motzkin word begins or ends with a parenthesis.

3.3. RNA folding

The SCFGs becomes useful in describing ribonucleic acid (RNA) secondary structure. Four types of nucleotides are the building blocks of RNA, which connect either "through hydrogen bonding or base stacking" [8]. These interactions occur in four basic secondary structural motifs: helices (also referred to as stacks or stems), hairpin loops, junctions (consisting of internal loops, and multi-loops), and bulges. These motifs create a "folding" phenomenon within RNA strands. The nucleotides also have interactions existing in tertiary structures, such as pseudoknots, which consist of overlapping structures. However, due to the stronger nature of the bonding energy in the secondary structure, the secondary and tertiary structural elements are separable; the secondary structure of RNA is stable and can exist on its own [8]. Therefore, we will neglect the tertiary structure when describing RNA folding.

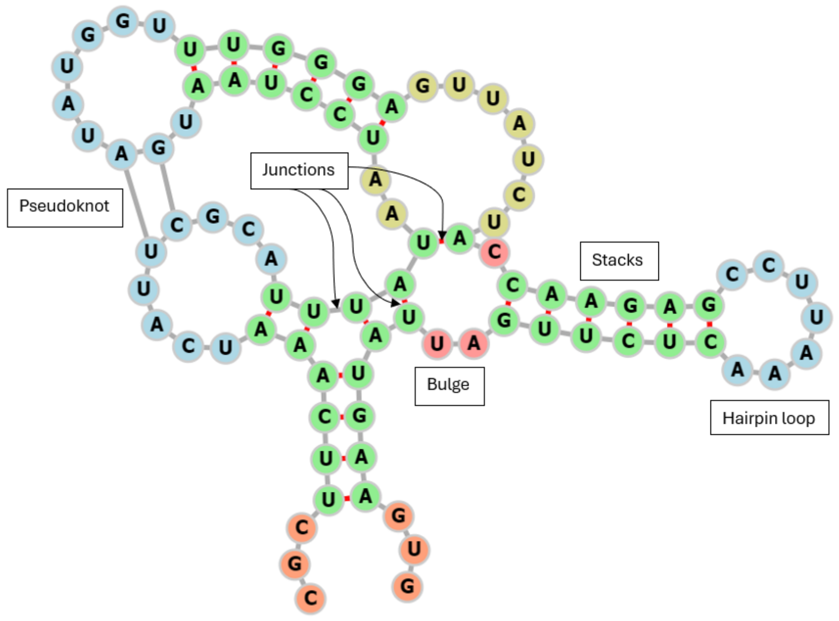

Figure 3.3. An example of the RNA secondary structure, created with [6], showing the motifs of stacks, hairpin loops, bulges, and junctions, as well as one pseudoknot. Pseudoknots form when motifs bond with each other. The Motzkin word for this RNA strand is "...((((((.........))(((..((((((.........))))))).......).((((((.......))))))..))))))...", removing the pseudoknot.

The structure of RNA consists mostly of base pairs, which form stacks. The four nucleotide bases of RNA are adenine, guanine, uracil, and cytosine, or A, G, U, and C, respectively. Adenine and guanine are purines, while uracil and cytosine are pyramidines. Due to the atomic structure of the nucleotides, purine-purine and pyramidine-pyramidine pairs are highly unlikely, as the molecules are, respectively, too close and too far apart for the hydrogen bonds to form. A-C pairs are also unlikely, as the hydrogen donors and acceptors do not agree. G-U pairs occur frequently as wobble base pairs, which give rise to tertiary structures such as pseudoknots. The only remaining base pairs are G-C and A-U. These base pairs form stacks, which provide structural stabilization for the RNA strands. However, when RNA folds, these stacks are interrupted by single strands of nucleotides or by other stacks, both of which form the hairpins, bulges, and different kinds of junctions, as seen in Figure 3.3. The bonding between nucleotides in multiple manners yields the RNA secondary structure that we will analyze.

3.3.1. Grammars, hidden Markov models, and RNA

There is a connection between RNA strands, SCFGs, and HMMs. HMMs are equivalent to a grammar called a stochastic regular grammar, where all production rules are of the form $W \to aW$ or $W \to a$. Regular grammars form strings from left to right and can be easily depicted in one-dimension. The terminals in a regular grammar represent the states of the HMM, the derived strings represent the state sequence, and the emission events are treated separately [2].

The one-dimensional nature of HMMs does not provide a mechanism to track pairs that are the base of the RNA structures. Instead, we can use the double emission normal form described in Section 3.1.1. Because the terminals "(" and ")" must be emitted simultaneously, we can easily represent and track base pairs of nucleotides. The terminal "." thus represents any unpaired nucleotide. We can then represent the secondary structure of RNA by Motzkin words. If we "unfold" the RNA structure and align the nucleotides in a line, the base pairs would be connected as in Figure 3.2, where no base pairs overlap. (As an aside: Though we neglect pseudoknots here, allowing of pseudoknots can lead to incorrect pairing of parentheses in the Motzkin word depiction from Figure 3.2. In Figure 3.3, if we do not remove the pseudoknot, the Motzkin word is "...((((((....((...))(((..((((((.))......)))))).......).((((((.......))))))..))))))...", which now has structural motifs incorrectly represented.).

While each terminal symbol represents a nucleotide, it is important to remember that each nucleotide is an A, G, U, or C. Thus, unfolding an RNA strand creates a sequence of letters. When looking at RNA, it is not always possible to see the structure itself. Instead, it is easier to read the sequence of nucleotides. This gives the RNA an HMM-esque quality; the hidden state sequence of is a sequence of terminal variables (or, in terms of the strand itself, we do not know whether the nucleotide is part of a base pair or is unpaired), and the observed events are the types (or "labels") of the nucleotides. Thus, within the SCFGs representing RNA folding, we also have emission probabilities for each production rule, which demonstrating how likely it is that a paired or unpaired nucleotides are each type.

As mentioned previously, the double emission normal form will be most convenient in describing RNA secondary structure. In general, structural motifs can be generated from sequences of production rules as follows [11]:

- Stacks: $V_1 \to (V_1)$

- Hairpins: $V_1 \to (V_2)$, $V_2 \to V_2V_3 \mid V_3V_3$, and $V_3 \to .$

- Bulges: $V_1 \to (V_1)$, $V_2 \to V_3V_1$, and $V_3 \to V_3V_3 \mid .$

for non-terminals V_i. While this is not always the case, it can be useful later when analyzing a parse tree to identify these sequences, and thus identify the potential structural motif associated to it.

3.3.2. RNA strand and grammars

Say we have a simple grammar defined by the production rules:

$$\begin{aligned} &S \to SS \mid LS \\ &L \to LL \end{aligned} \tag{3.8}$$

where $S \to SS$ has probability 0.3 and $S \to LS$ has probability 0.7. For the sake of simplicity, we will neglect the productions $S \to .$ and $L \to .$, replacing them with the following emissions:

$$e_S(A) = e_S(C) = 0.4, \quad e_S(U) = e_S(G) = 0.1 \tag{3.9}$$

and

$$e_L(A) = e_L(C) = e_L(U) = e_L(G) = 0.25. \tag{3.10}$$

Now, for an observation of the string $ACCUG$, we may have the following parse tree:

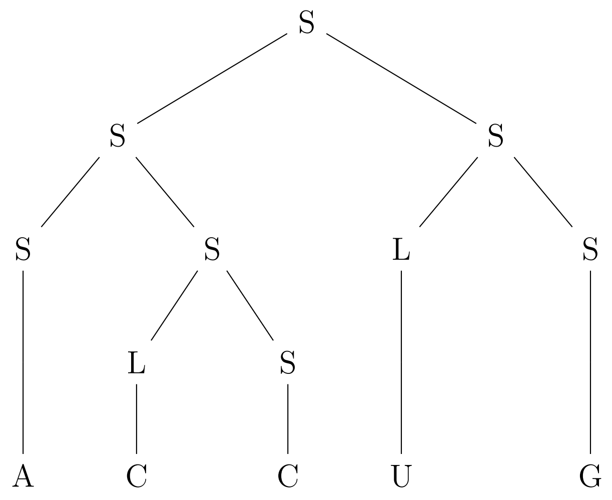

Figure 3.4. Depicted is a sample parse tree for the string *ACCUG*, generated from production rules in Equation 3.8.

We can find the probability of this parse tree emitting the strand $ACCUG$ by multiplying all production and emission probabilities:

$$\begin{aligned} P[S \to SS]^2 \cdot P[S \to LS]^2 \cdot e_S(A) \cdot e_S(C) \cdot e_S(G) \cdot e_L(C) \cdot e_L(U) \\ = 0.3^2 \cdot 0.7^2 \cdot 0.4^2 \cdot 0.25^2 \cdot 0.1 \\ = 0.0000441. \end{aligned} \tag{3.11}$$

From the parse tree in Figure 3.4, we can also see which substrings derived from which production rules. For example, in the first production rule $S \to SS$, we can see that the substring ACC derived from the left S, while the substring UG derived from the right S. This sort of tracking will be useful later in Section 3.4.

3.4. Cocke-Younger-Kasami algorithm

The Cocke-Younger-Kasami (CYK) algorithm provides a way to find determine which parse tree generated a specific sequence within a given SCFG. Analogous to the Viterbi algorithm for HMMs, the CYK algorithm finds the most probable parse tree. The CYK algorithm breaks a given string down into smaller substrings, finding the most probably parse tree for that substring, growing the substring by one element in each step, and continuing until the full string as been achieved.

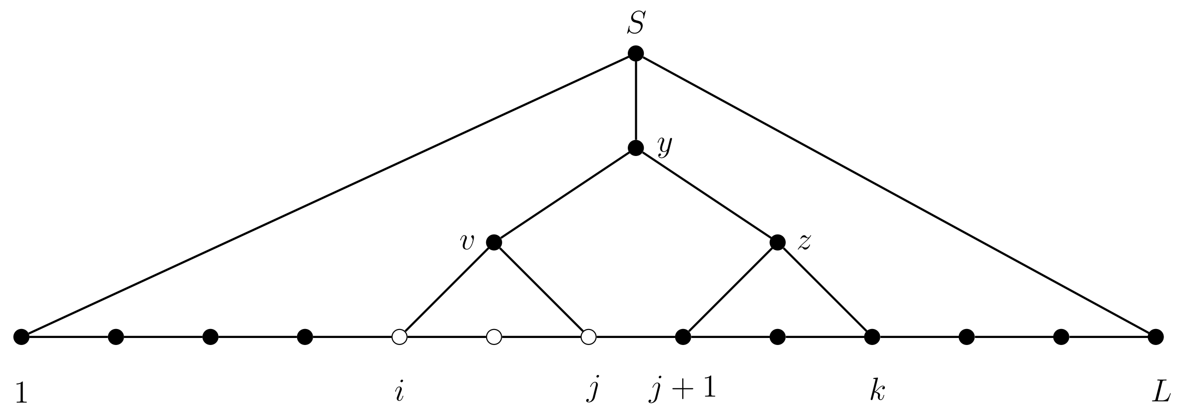

Figure 3.5. Shown is a depiction of the CYK algorithm. The CYK algorithm considers fragments of an observed string from x_i to x_k and finds the most probable breaking point j. The most probable breaking point is determined by finding the most probable outcome from the production rule and emission probabilities. With the most probable breaking point, we also track which production rule created it, shown as $y \to vz$. The algorithm is repeated until the entire string, from x_1 to x_L has been considered, noting that the string always emerges from start symbol S.

The CYK algorithm utilizes a variable $\gamma(i, k, y)$, which is the most probable parse tree for the string beginning at i and ending at k with root y, shown in Figure 3.5. The probability of γ is denoted $P(x|\gamma)$ for observed string x. A traceback mechanism is also necessary to recover the most probable structure. As we determine the most probable tree for each substring, we create the traceback table simultaneously, recording the double $(y \to vz, j)$ to denote that the most probable tree was generated from production rule $y \to vz$ with breaking point j. In the traceback table, we assume the start symbol is S; thus, for entry (i, i) along the diagonal, we have (S, i) for all i.

3.4.1. An example

We consider the same simple grammar as in Section 3.3.2 with observed string $ACCUG$, where production rules $S \to SS$ and $S \to LS$ occurred with probability 0.3 and 0.7, respectively. This time, however, we do not know the associated parse tree. We will use the CYK algorithm to find the most probable one.

The CYK algorithm requires tracking the most probable outcome for both production rules S and L, iterating along the length of the substrings. Along the diagonals in Table 3.3, we see the probability of a string of length 1 emitting each corresponding symbol from both S and L. Then, we consider strings of length 2, working adjacently above the main diagonal. Because L only has the production rule $L \to LL$, the L table consists of multiplying emission probabilities for the respective substrings. Thus, because $e_L(x_i) = e_L(x_j) = 0.25$ for all i and j, a substring of length n has probability 0.25^n is the L table.

Then, for the S table, we must compare and determine whether the strings of length 2 were more likely to come from the production rule $S \to SS$ or $S \to LS$. For entry (i, j), we have the substring AC, which either came from the production LS or SS. Using the probabilities already determined along the diagonals, we have the following possibilities:

$$A - C \text{ came from } S - S : 0.3 \cdot 0.4 \cdot 0.4 = 0.048 \tag{3.12}$$

$$A - C \text{ came from } L - S : 0.7 \cdot 0.25 \cdot 0.4 = 0.07. \tag{3.13}$$

where we denote the break in the substring with a hyphen. Here, AC is more likely to be emitted from the production rule LS. In our traceback table, our break point is 1, we record

$(S \to LS, 1)$ as A emitted from L and C emitted from S, and in our S table, we record the value $\gamma(1, 2, S \to LS) = 0.07$, for entries $(1, 2)$.

The same comparison is made for all substrings of length 2. The process then repeats for length 3. As the substrings grow, more comparisons are to be made. As another example, consider the entry $(2, 5)$, for which we have the substring $CCUG$. Because we have already calculated values for strings of length 1, 2, and 3, we are able to consider all possible outcomes for the string of length 4:

$$C - CUG \text{ came from } S - S : 0.3 \cdot 0.4 \cdot 0.1 = 0.0021 \tag{3.14}$$

$$CC - UG \text{ came from } S - S : 0.3 \cdot 0.07 \cdot 0.0175 = 0.0003675 \tag{3.15}$$

$$CCU - G \text{ came from } S - S : 0.3 \cdot 0.004375 \cdot 0.0175 = 0.0013125 \tag{3.16}$$

$$C - CUG \text{ came from } L - S : 0.7 \cdot 0.25 \cdot 0.0175 = 0.0030635 \tag{3.17}$$

$$CC - UG \text{ came from } L - S : 0.7 \cdot 0.0625 \cdot 0.0175 = 0.000765625 \tag{3.18}$$

$$CCU - G \text{ came from } L - S : 0.7 \cdot 0.015265 \cdot 0.1 = 0.00109375. \tag{3.19}$$

Thus, the $(2, 5)$ entry in the S table (Table 3.1) is 0.0030625 and in the traceback table is $(S \to LS, 2)$, as $x_2 = C$ is the breaking point. The full tables of values are shown below in Table 3.1, Table 3.2 and Table 3.3.

Continuing this process, we determine the remaining entries in each table. Then, we must find the most probable parse tree. In the traceback table in Table 3.3, we first look at the entry $(1, 5)$, which corresponds to our full string $ACCUG$. The entry of $(S \to LS, 2)$

	A	C	C	U	G
A	0.4	0.07	0.0175	0.001060855	0.000765625
C		0.4	0.07	0.004375	0.0030625
C			0.4	0.0175	0.0175
U				0.1	0.0175
G					0.1

Table 3.1. The S table for the example CYK algorithm.

	A	C	C	U	G
A	0.25	0.0625	0.015265	0.00390625	0.0009765625
C		0.25	0.0625	0.015265	0.00390625
C			0.25	0.0625	0.015265
U				0.25	0.0625
G					0.25

Table 3.2. The L table for the example CYK algorithm.

	A	C	C	U	G
A	$(S,1)$	$(S \to LS,1)$	$(S \to LS,2)$	$(S \to LS,3)$	$(S \to LS,2)$
C		$(S,2)$	$(S \to LS,2)$	$(S \to LS,3)$	$(S \to LS,2)$
C			$(S,3)$	$(S \to LS,3)$	$(S \to LS,3)$
U				$(S,4)$	$(S \to LS,4)$
G					$(S,5)$

Table 3.3. The traceback table for the example CYK algorithm.

tells us that AC emitted from L, and CUG emitted from S. Thus, from root S, we have production LS.

Then, because AC emitted from L and L has only the production rule $L \to LL$, $A - C$ must have emitted from $L - L$. For CUG, we must use our traceback entry $(3,5)$, where we see $C - UG$ emitted from $L - S$. Similarly, we see the final traceback for $U - G$ is $L - S$.

The traceback process then tells us that the most likely parse tree for $ACCUG$ is the tree depicted in Figure 3.6 with probability 0.00765625.

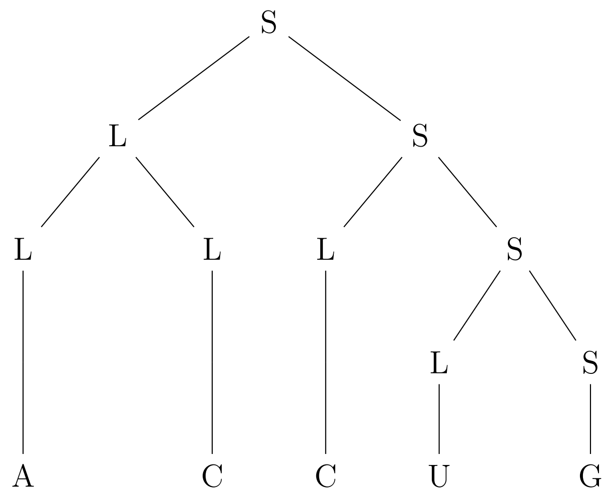

Figure 3.6. The most probable parse tree for $ACCUG$ came from the derivation rules, from top-to-bottom, left-to-right, $S \to LS$, $L \to LL$, $S \to LS$, and $S \to LS$.

3.4.2. KH99' and other grammars

The following analysis is summarized from [11].

The CYK algorithm was used to measure the accuracy of grammars in predicting the structure of RNA strands. Two of these grammars developed as variations of the KH99' grammar using the evolutionary algorithm. For the sake of consistency, the rule $B \to .$ was kept throughout. The production rules for KH99' and two of its variations are described in Table 3.4, as well as their PPV and sensitivity values. GG1 is KH99' with two added rules, which explains the similarity in accuracy under the CYK algorithm. GG2 has almost all possible rules, which lends itself to more ambiguity, and thus performs worse comparatively in both measurements. It is worth noting that while SCFGs are able to predict RNA secondary structure, the low PPV and sensitivity values show that there is still much more work to do to improve the quality of the predictions made. The SCFGs evolve and improve through a process called the *evolutionary algorithm*.

KH99’	$A \to BA \vert . \vert (C)$ $B \to . \vert (C)$ $C \to BA \vert (C)$	PPV: 0.479 Sensitivity: 0.496
GG1	$A \to AA \vert BA \vert . \vert (A) \vert (C)$ $B \to . \vert (C)$ $C \to BA \vert (C)$	PPV: 0.481 Sensitivity: 0.505
GG2	$A \to AA \vert AB \vert BA \vert BB \vert CB \vert CB \vert . \vert (B) \vert (C)$ $B \to .$ $C \to AA \vert AB \vert BA \vert BB \vert BC \vert CA \vert CB \vert . \vert (A) \vert (B) \vert (C)$	PPV: 0.258 Sensitivity: 0.330

Table 3.4. The rules for the KH99’ grammar and two of its variations, as well as their PPV and sensitivity scores, found using the CYK algorithm.

While SCFGs can predict RNA secondary structure, as of current, they are an alternative to the more commonly used energy models. These energy models consist of lab experiments to determine the amount of energy required to break bonds in a segment of RNA strand, which then can be compared to accepted minimum values for each motif. SCFGs, on the other hand, are easier to handle combinatorially and require fewer parameters, potentially making them more appealing. However, as seen in Table 3.4, the accuracy measures are low, and more work must be done to improve the grammars to make them consistently good.

3.4.3. Evolutionary algorithm

The goal of using SCFGs here is to find the grammars that best represent and predict RNA secondary structure. Using the double emission normal form with m non-terminal variables, we have m^3 possible production rules of the form $T \to UV$, m^2 possible rules of the form $T \to (U)$, and m possible rules of the form $T \to .$, yielding $2^{m^3+m^2+m}$ possible grammars. To allow for efficient search of the space of grammars in such form, an evolutionary algorithm must be well-designed. The algorithm searches for grammars by first appropriately designing of an initial population of production rules and later utilizing mutation, breeding, and selection procedures [11]. The grammars GG1 and GG2 derived from KH99' using the evolutionary algorithm.

Choosing the initial population depends on the size of the space of grammars. Even for small values of m, the population size is too large to approach with an evolutionary algorithm. Thus, it is practical to start with an initial population of grammars consisting of two non-terminals, and later use mutation and breeding methods to expand the number of non-terminals and production rules. With two non-terminals S and T, we have possible production rules

$$\begin{aligned} S &\rightarrow \left\{ \begin{array}{c|c|c|c|c|} SS & ST & TS & TT & \\ & & & & (S) \\ -- & -- & -- & -- & \end{array} \right. . \\ T &\rightarrow \ . \end{aligned} \tag{3.20}$$

where sixteen total combinations can be made by including and excluding any of the four rules of the form $S \rightarrow UV$ and always including the remaining rules.

After designing the initial population of grammars, the evolutionary algorithm can begin. Mutations lead the majority of the search through the space of grammars, as they create more structural freedom from the initial population. For non-terminals V_i in $\mathcal{N}$, the mutation rules used are:

- The start variable changes,
- A production rule is added or deleted,
- A new non-terminal V' is added, along with two new production rules to ensure that it is reachable and is indeed non-terminal (such that $P_{V'} \neq \emptyset$ or $\varnothing$),

- A new non-terminal variable V' is created with rules identical to a pre-existing one,
- A production rule of the form $V_i \to V_j V_k$ is changed to $V_i \to V_j V_l$, $V_i \to V_l V_k$, or $V_i \to V_l V_p$,
- A production rule of the form $V_i \to (V_j)$ is changed to $V_i \to (V_k)$.

The forms of mutation here are rather basic. As a search through a space of grammars is the goal, the rate of mutation must be done wisely. Adding rules too slowly does not allow the grammar develop structure, while adding too quickly results in redundancy. More complex mutations are certainly possible, but are not needed; a search through the given space of grammars is sufficient with only the mutations listed. It is also worth noting that deleting rules almost always resulted in a worse grammar, where the accuracy of predictions decreased.

In the breeding model, a new grammar G is produced from breeding two previously introduced grammars G_1 and G_2, which have terminals "$($", "$)$", and "$.$" and non-terminals U_1, U_2, ..., U_n and V_1, V_2, ..., V_m, respectively. The new grammar G is given start symbol S and the production rules:

- $P_S = P_{S_1} \cup P_{S_2}$,
- P_{U_i}, where all instances of S_1 are replaced with S, and
- P_{V_i}, where all instances of S_2 are replaces with S.

In other words, the breeding procedure consists of merging two grammars into one, keeping all production rules from both, but replacing the original start symbols S_1 and S_2 with one single non-terminal start symbol S.

In the evolutionary algorithm, because of the large space of grammars being used, selection must be utilized to determine the most accurate grammars. The SCFGs are used to predict RNA secondary structure, and thus can be analyzing using the accuracy measurements of PPV and sensitivity described earlier. In general, the most accurate grammars (with the highest PPV and sensitivity values) are kept. It is also useful to keep less accurate grammars, as well, as the mutation and breeding process can be repeated in attempt to improve those grammars.

3.4.4. Ambiguity and completeness

In the evolutionary algorithm, there are a couple of factors to consider: ambiguity and completeness. A grammar is *ambiguous* if it produces more than one derivation for a given structure (string). Because we are interested in representing RNA structures using SCFGs, we need to consider the importance of ambiguity. Ambiguous grammar in this context are small, with at most two non-terminal variables, and perform relatively poorly compared to the unambiguous ones. However, the poor predictive quality is likely due to design deficiencies rather than ambiguity, and thus ambiguous grammars are not necessarily undesirable [11].

In the context of RNA secondary structure, a grammar is said to be *complete* "if it has a derivation for all possible structures which have no hairpins shorter than length two" [11]. While it is desirable for a grammar to be complete and to be able to generate all structures, it is likely that many structures have probabilities near zero. Thus, completeness is also difficult to ensure.

In application, a complete, unambiguous grammar cannot be modified without compromising either of the properties. It is very difficult to ensure both qualities are met. In general, adding production rules creates ambiguity, while removing production rules creates incompleteness. Thus, grammars considered here include grammars that are either ambiguous, incomplete, or both.

REFERENCES

[1] H. Bunke, M. Roth, and E.G. Schukat-Talamazzini. Off-line cursive handwriting recognition using hidden markov models. *Pattern Recognition*, 28(9):1399–1413, 1995.

[2] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids.* Cambridge University Press, 1998.

[3] Sean R Eddy. What is a hidden markov model? *Nature Biotechnology*, 22(10):1315–1316, October 2004.

[4] K.-F. Lee and H.-W. Hon. Large-vocabulary speaker-independent continuous speech recognition using hmm. In *ICASSP-88., International Conference on Acoustics, Speech, and Signal Processing*, pages 123–126 vol.1, 1988.

[5] S. E. Levinson, L. R. Rabiner, and M. M. Sondhi. An introduction to the application of the theory of probabilistic functions of a markov process to automatic speech recognition. *The Bell System Technical Journal*, 62(4):1035–1074, 1983.

[6] Ronny Lorenz, Stephan H Bernhart, Christian Hoener Zu Siederdissen, Hakim Tafer, Christoph Flamm, Peter F Stadler, and Ivo L Hofacker. Viennarna package 2.0. *Algorithms for Molecular Biology*, 6(1):26, 2011.

[7] Nguyet Nguyen. Hidden markov model for stock trading. *International Journal of Financial Studies*, 6(2):36, March 2018.

[8] Ignacio Tinoco and Carlos Bustamante. How rna folds. *Journal of Molecular Biology*, 293(2):271–281, 1999.

[9] Robert Trevethan. Sensitivity, specificity, and predictive values: Foundations, pliabilities, and pitfalls in research and practice. *Frontiers in Public Health*, 5, November 2017.

[10] Philip V'kovski, Annika Kratzel, Silvio Steiner, Hanspeter Stalder, and Volker Thiel. Coronavirus biology and replication: implications for sars-cov-2. *Nature Reviews Microbiology*, 19(3):155–170, 2021.

[11] James WJ Anderson, Paula Tataru, Joe Staines, Jotun Hein, and Rune Lyngsø. Evolving stochastic context-free grammars for rna secondary structure prediction. *BMC Bioinformatics*, 13(1), May 2012.

APPENDIX

A. Code

Here is the code for the dishonest casino, with variations used to run samples in Section 2.2.1, Section 2.4.1, Section 2.5.1, and Section 2.6.

```
import random


sens = []

pos_pred = []

n = 0

j = 0

trans_mat = [[0.5, 0.5], [0.8, 0.2], [0.3, 0.7]]

emis_mat = [[1/6, 1/6, 1/6, 1/6, 1/6, 1/6], [0.1, 0.1, 0.1, 0.1, 0.1, 0.5]]


while j < 1000:

    S = ''

    R = ''

    fair = [1,2,3,4,5,6]

    loaded = [1,2,3,4,5,6,6,6,6,6]


    for i in range(200):

        if i == 0:

            num = random.random()

            if num < 0.5:
```

```
        S = S + "F"

        roll = random.choice(fair)

        R = R + str(roll)

    if num >= 0.5:

        S = S + "L"

        roll = random.choice(loaded)

        R = R + str(roll)

if i != 0:

    num = random.random()

    if S[-1] == "F":

        if num < trans_mat[1][0]:

            S = S + "F"

            roll = random.choice(fair)

            R = R + str(roll)

        if num >= trans_mat[1][0]:

            S = S + "L"

            roll = random.choice(loaded)

            R = R + str(roll)


    else:

       if num < trans_mat[2][0]:

           S = S + "F"

           roll = random.choice(fair)
```

```
                R = R + str(roll)


            if num >= trans_mat[2][0]:

                S = S + "L"

                roll = random.choice(loaded)

                R = R + str(roll)


    rolls = R


    vF = 0

    vL = 0

    Fdie = trans_mat[1]

    Ldie = trans_mat[2]

    F = emis_mat[0]

    L = emis_mat[1]

    die = 1

    vList = [[], []]


## step 1

    vList[0].append(die*trans_mat[0][0]*F[0])

    vList[1].append(die*trans_mat[0][1]*L[0])

    FLlist = [["0"], ["0"]]
```

```
## iterate over remaining values

    for i in rolls[1:]:
        FF = vList[0][-1]*Fdie[0]*F[int(i)-1]
        FL = vList[0][-1]*Fdie[1]*L[int(i)-1]

        LF = vList[1][-1]*Ldie[0]*F[int(i)-1]
        LL = vList[1][-1]*Ldie[1]*L[int(i)-1]

        if FF > LF:
            vList[0].append(FF)
            FLlist[0].append("F")
        if LF >= FF:
            vList[0].append(LF)
            FLlist[0].append("L")

        if FL > LL:
            vList[1].append(FL)
            FLlist[1].append("F")
        if LL >= FL:
            vList[1].append(LL)
            FLlist[1].append("L")
```

```
guess = ’’


if vList[0][-1] >= vList[1][-1]:

    guess = "F" + guess

if vList[1][-1] > vList[0][-1]:

    guess = "L" + guess


for i in range(1,len(rolls)):

    if guess[0] == ’F’:

        guess = str(FLlist[0][-i]) + guess


    else:

        guess = str(FLlist[1][-i]) + guess


nL = S.count("L")

nLG = guess.count("L")


for i in range(len(rolls)):

    if S[i] == guess[i] and S[i] == "L":

        n = n+1


if nL != 0 and nLG != 0:

    sens.append(n/nL)
```

```
        pos_pred.append(n/nLG)


    else:

        sens.append(100)

        pos_pred.append(100)


    #print("Guess = ", guess)

    #print("Actual = ", S)

    n = 0

    j = j+1

#print("Sensitivity = ", sens)

#print("Positive-Predictive = ", pos_pred)
```

B. Comparisons

Here are the remaining calculated values used for comparison to generate the S table in the example of the CYK algorithm in Section 3.4.1.

Length 2:

CC:

$$C - C \text{ came from } S - S : 0.048$$

$$C - C \text{ came from } L - S : 0.07$$

CU:

$$C - U \text{ came from } S - S : 0.012$$

$$C - U \text{ came from } L - S : 0.0175$$

UG:

$$U - G \text{ came from } S - S : 0.003$$

$$U - G \text{ came from } L - S : 0.0175$$

Length 3:

ACC:

$$A - CC \text{ came from } S - S : 0.0084$$

$$AC - C \text{ came from } S - S : 0.0084$$

$$A - CC \text{ came from } L - S : 0.01225$$

$$AC - C \text{ came from } L - S : 0.0175$$

CCU:

$C - CU$ came from $S - S$: 0.0021

$CC - U$ came from $S - S$: 0.0021

$C - CU$ came from $L - S$: 0.0030625

$CC - U$ came from $L - S$: 0.004375

CUG:

$C - UG$ came from $S - S$: 0.0021

$CU - G$ came from $S - S$: 0.000525

$C - UG$ came from $L - S$: 0.0175

$CU - G$ came from $L - S$: 0.0109

Length 4:

$ACCU$:

$A - CCU$ came from $S - S$: 0.000525

$AC - CU$ came from $S - S$: 0.0003675

$ACC - U$ came from $S - S$: 0.000525

$A - CCU$ came from $L - S$: 0.000765625

$AC - CU$ came from $L - S$: 0.000765625

$ACC - U$ came from $L - S$: 0.00106855

Length 5:

$ACCUG$:

$A - CCUG$ came from $S - S$: 0.0003675

$AC - CUG$ came from $S - S$: 0.0003675

$ACC - UG$ came from $S - S$: 0.000091875

$ACCU - G$ came from $S - S$: 0.0000320565

$A - CCUG$ came from $L - S$: 0.0005359375

$AC - CUG$ came from $L - S$: 0.000765625

$ACC - UG$ came from $L - S$: 0.00018699625

$ACCU - G$ came from $L - S$: 0.0002734375
