Implementing and Comparing RNA Secondary Structure Prediction Algorithms

Ethan Hersch[1], **Ben Bigdelle**[2], **Malli Gutta**[3], **and Mark Beckmann**[4]

[1]Computer Science & Mathematics, 2026, esh87
[2]Computer Science, 2025, brb227
[3]Mathematics, 2026, mg2395
[4]Computer Science, 2025, mcb373

ABSTRACT

RNA secondary structure prediction is a computationally expensive task that has wide use-cases in biological fields and research. This project explores RNA secondary structure prediction by implementing and comparing algorithms, including the Nussinov algorithm, its optimized and Four Russians versions, the Zuker algorithm, and a novel convolutional neural network (CNN) approach. These methods are evaluated for runtime efficiency and prediction accuracy using Levenshtein distance against over 500 RNA sequences. While the Nussinov algorithm provides a foundational baseline from the textbook, optimizations and biologically motivated approaches like Zuker improve prediction fidelity by minimizing free energy. The CNN introduces a machine learning perspective, transforming prediction into a classification problem and demonstrating potential in handling complex sequences. This project highlights trade-offs in computational cost, accuracy, and biological relevance, offering insights into advancing RNA folding prediction and fostering innovation in computational biology.

Keywords (minumum 5): RNA folding, stack programming, dynamic programming, Nussinov Algorithm, energy minimization, secondary structure prediction, optimization, benchmarking, neural network

Project type: Reimplementation / Benchmarking

Project repository: https://github.coecis.cornell.edu/mcb373/CS4775_Project

Main dataset: https://zenodo.org/records/7788537

1 Introduction

Understanding RNA folding is crucial for deciphering cellular functions, as the folded structure of RNA molecules determines their ability to execute vital biological roles. RNA molecules are involved in processes such as protein synthesis, gene regulation, and catalysis. However, for RNA to be functional, it must first fold into specific secondary and tertiary structures. Predicting RNA folding patterns helps us understand how RNA achieves functionality, which has broad implications for biology and medicine, such as designing RNA-based therapeutics and understanding disease mechanisms [1, 2].

RNA is a single-stranded molecule composed of nucleotides, each containing one of four bases: adenine (A), uracil (U), guanine (G), or cytosine (C). RNA folds into complex structures due to its bases pairing, where A pairs with U and G pairs with C through hydrogen bonds. In RNA secondary structures, however, G can pair with U to form wobble pairings. During folding, the RNA strand forms secondary structures, such as stems (paired regions) and loops (unpaired regions). Loops occur when base pairing is interrupted, creating structures like hairpins (a stem followed by an unpaired region), bulges, or internal loops. These loops are important because they add free energy to the molecule—unpaired bases are less stable than paired ones. Balancing these paired and unpaired regions helps RNA adopt its functional form with minimal free energy as this leads to more thermodynamic stability [1, 2].

In this project, we aim to predict RNA secondary structures using various algorithms and evaluate their efficiency and accuracy. Specifically, we are investigating the Nussinov algorithm, an optimized variant, the Four-Russians implementation of the Nussinov algorithm, the Zuker algorithm, and a neural network approach. The need for this level of depth arises from the high time complexity of current RNA folding algorithms. Identifying optimizations that reduce computational costs while maintaining prediction accuracy could have significant implications in computational biology [3–5].

The Nussinov algorithm, a foundational method for RNA secondary structure prediction, operates by identifying the optimal pairing of nucleotides to maximize the number of stable base pairs. Using dynamic programming, it builds a matrix where each cell represents the maximum number of base pairs possible in specific subsequences of the RNA. The algorithm recursively computes these values and backtracks to determine the RNA structure. While it is moderately effective, the Nussinov algorithm has a time complexity of $O(n^3)$ and a space complexity of $O(n^2)$, limiting its scalability for longer RNA sequences [6].

Our project will initially implement the Nussinov algorithm as described in Chapter 10 of the Durbin textbook [7]. We will explore potential optimizations to improve its efficiency in time and memory usage, leveraging algorithmic improvements. One such optimization, the Four-Russians method, uses precomputed blocks to reduce computational complexity, which we plan to evaluate for its practical performance gains [8, 9]. We also aim to benchmark the Zuker algorithm, a more biologically accurate RNA folding prediction method that balances thermodynamic stability with structural constraints.

We test accuracy and speed on over 500 RNA sequences with their secondary structures found from an online Zenodo dataset [10]. This data was affiliated with Poznan University of Technology's Institute of Computing Science. It turns out its main purpose is to benchmarking ML approaches in RNA 2D structure prediction tasks.

To this end, we utilize standard data science and visualization tools to evaluate performance of our algorithms and propose which methodologies work best. This involves fitting linear regression models to our datasets collected on these 500+ sequences to see how specific methods exhibit more or less variance compared to others. We also introduce data analysis techniques to observe if certain algorithms favor pairing specific base pairs more than others. Later, we experiment with training a neural network for RNA secondary structure prediction, given we had immense amounts of training data of various sizes.

On top of using data-oriented approaches to display the tradeoffs of specific algorithms, we implemented a graphical user interface (GUI) so users can see how the physical structure of an RNA sequence compares when folded with different algorithms. The user can even experiment with making their own alterations of the algorithms. For example, we propose two methods of traceback for the Nussinov algorithm and allow the user to experiment with different score matrices for the Zuker algorithm and they are even allowed to create their own score matrix when interacting with our command line.

Ultimately, this project serves as both an implementation study and a benchmarking comparison of RNA folding algorithms. By focusing on the Nussinov algorithm and its optimized versions, we aim to advance our understanding of RNA folding prediction and contribute to the broader field of computational biology. We also benchmark and implement other promising approaches to RNA folding, such as the Four-Russians optimization and using deep learning. Integrating machine learning approaches, as described in recent studies [11, 12], could improve predictions for complex structures or sequences. These avenues of exploration will be noted in our discussion of future work.

2 Methods

The core of our project is the multiple algorithms we implement to evaluate performance on our dataset. We first implemented the fundamental algorithms of RNA folding, Nussinov and Zuker, and then experienced with various optimizations and improved algorithms. After implementing these algorithms correctly, we introduced both qualitative and quantitative evaluation metrics for our algorithms. The quantitative ones include accuracy to the correct secondary structure, using Levenshtein distance, and runtime for executing each of these algorithms. Our codebase provides terminal commands so the user can see these tradeoffs between algorithms on the baseline dataset as well as any sequence they want to input with a visualization via a GUI. We then use basic data science and analysis tools to provide interpretable results directly showing the tradeoffs of each algorithm and these results are discussed more later. The qualitative results include the GUI implemented in our codebase so users can see exactly how structures may differ for any sequence they enter.

2.1 Dot-Bracket Notation

Dot-bracket notation is a simplified representation of RNA secondary structures. In this format, unpaired nucleotides are denoted by dots ('.'), and base pairs are represented by matching parentheses: opening pairs as '(' and closing pairs as ')'. For example, the sequence `AUGCUAG` with a stem-loop structure might be represented as `(((...)))`, indicating three base pairs enclosing an unpaired loop. For example if 'AAUCGU' has index 1 paired to 6, 4 to 5, this becomes '(. . ( ))' and this is the structure we use to represent secondary structure.

This notation provides a compact way to predict and visualize RNA folding, and this is the way that our predictions were output for all algorithms and methods. We evaluate how "good" a prediction is by taking the Levenshtein distance between our prediction and the true value.

2.2 Nussinov Algorithm

The Nussinov algorithm is the first algorithm we implemented as it is a rudimentary dynamic programming algorithm for naive secondary structure prediction. While it rarely mimics biological behavior, its simplicity and intuition make it a good baseline to build more complex algorithms that rely on energy calculations. The core principle of the algorithm is to identify pairs of bases in the sequence that can form valid base pairs (e.g. A-U, G-C, G-U), while maximizing the total number of such pairs. Note, G and U can now pair: this is called a wobble base pair and is a key difference with RNA folding which does not just consider Watson-Crick pairs [13]. The algorithm employs a recursive relationship, systematically breaking the RNA sequence into smaller subproblems and combining solutions to solve the entire problem (this process sped up using dynamic programming with a 2D DP table).

See [4.1] for the written algorithm. First, initialize a 2D DP table with zeros on the main diagonal and diagonal below it. Also define a score function where base pairs A-U, G-U, and C-G have value 1, 0 otherwise. Say S is our input string representing the RNA sequence. Then construct the recurrence relation: given a subsequence $S[i:j]$, determine $DP[i][j]$. Leaving base i unpaired returns $DP[i+1][j]$. Leaving base j unpaired returns $DP[i][j-1]$ first. Pairing base i with j returns $DP[i+1][j-1] + pair(S[i], S[j])$. But another option is to pair with some index $k \in (i, j)$, resulting in $max_{i<k<j}(pair(S[i], S[k]) + DP[k+1][j])$. Thus the final recurrence is

$$DP[i][j] = \max \begin{cases} DP[i+1][j], & \text{(base } i \text{ is unpaired)} \\ DP[i][j-1], & \text{(base } j \text{ is unpaired)} \\ DP[i+1][j-1] + \text{pair}(S[i], S[j]), & \text{(bases } i \text{ and } j \text{ are paired)} \\ \max\limits_{i<k<j} \left(pair(S[i], S[k]) + DP[k+1][j] \right), & \text{(split at some } k\text{)} \end{cases}$$

We implement this using a bottom-up approach using for loops resulting in $O(n^3)$ time complexity and $O(n^2)$ space. This involves building up a DP score matrix; next we perform traceback on this to return the actual prediction. This involves building up a DP score matrix; next, we perform traceback on this to return the actual prediction. This is a stack-based approach which first involves pushing the indices of the top-right corner of our DP table onto a stack. Now we recurse on the following process until the stack is empty. Pop off this stack and if the row index (i) is at least the column index (j), continue this recursion. Otherwise, consider four cases:

1. **First Case:** If $DP[i+1][j]$ equals $DP[i][j]$, push $(i+1, j)$ onto the stack because this indicates that base i is unpaired in the optimal alignment. Thus, the optimal solution for the subsequence from i to j comes from the subsequence from $i+1$ to j, with no base pair involving i.

2. **Second Case:** If $DP[i][j-1]$ equals $DP[i][j]$, push $(i, j-1)$ onto the stack because this indicates that base j is unpaired in the optimal alignment. In this case, the optimal solution for the subsequence from i to j is identical to that for the subsequence from i to $j-1$, without pairing base j.

3. **Third Case:** If $DP[i+1][j-1] + \text{pair}(S[i], S[j])$ equals $DP[i][j]$, there must be an (i, j) base pair. This condition implies that the optimal solution includes pairing base i with base j, so we record this base pair and then push $(i+1, j-1)$ onto the stack. This corresponds to progressing in the DP table by skipping over the paired bases i and j.

4. **Fourth Case:** For $k = i+1$ to $j-1$, if $DP[i][j] = DP[i][k] + DP[k+1][j]$, this means the optimal solution for the subsequence from i to j involves splitting the sequence into two subsequences, one from i to k and the other from $k+1$ to j. In this case, push both (i,k) and $(k+1, j)$ onto the stack because the traceback process needs to be repeated for both of these subsequences.

This pseudocode can be found in the section Algorithms section 4.2. After this process, paired sequences are stored in a list of tuples. Thus we start with a string of the character '.' repeated L times (where L is the length of S) and each (i, j) pair in this list representing paired bases gets an open bracket '(' and closed bracket ')' respectively. This is to show our secondary structure in dot-bracket notation. In dot-bracket form, unpaired regions are dots and the bases which are paired to each other are represented as a set of parentheses [14]. The theory and explanation behind this algorithm was found in Durbin chapter 10.2 [7].

There is also a second form of performing traceback that also yields optimal structure and its algorithm can be found in section 4.3. The first Nussinov traceback algorithm identifies base pairs by recursively checking if bases i and j can pair directly or if splitting into subintervals maximizes the pairing score. It systematically explores all possible splits. The second algorithm, however, focuses on finding valid pairs by iterating over potential pairing indices k for i and j, prioritizing pairing constraints directly within the loop. This approach more explicitly integrates the $\text{pair}(S[i], S[j])$ check into the loop logic, reducing redundancy in splitting decisions [15]. Both are valid, and we give the user the option to compare both of these in the codebase.

2.3 Optimized Nussinov Algorithm

We first saught to make slight moderations to the Nussinov algorithm using rolling DP tables and precomputing scores for base pairs. This pseudocode can be found in section [4.4]. The traceback method is the same as described in the last section. This approach mainly reduces space, yet we found it can slightly improve time, depending on the dataset. This was a naive data-structures-oriented approach to optimization as a first attempt. Its results will be discussed later.

2.4 Zuker Algorithm

The Zuker algorithm is a dynamic programming-based approach used for predicting the secondary structure of RNA molecules. It aims to find the most thermodynamically stable configuration by minimizing the free energy of the RNA's secondary structure. The algorithm considers base-pairing interactions, loops, and various structural constraints to calculate the minimum free energy (MFE). Where the Nussinov algorithm was a purely computational approach and did not consider biological motivations of folding, the Zuker algorithm mimics more closely how folding occurs in real life by considering MFE [7]. We outline pseudo code of our algorithm in section [4.5].

We will outine the key steps of our algorithm. Define a table `MFE[i][j]` where i and j represent the indices of the RNA sequence. This table stores the minimum free energy of the subsequence from i to j. Compute the MFE based on possible structural elements:

- *Stacking pairs:* If bases i and j can pair, compute the energy contribution and add the value from the inner subsequence.
- *Hairpin loop:* Calculate the energy contribution of forming a hairpin loop between i and j.
- *Bulge/Internal loop:* Account for unpaired bases between paired segments.
- *Multi-branch loops:* Split the subsequence into two or more branches and calculate the energy recursively.

Reconstruct the secondary structure by tracing back through the dynamic programming table. Return the optimal secondary structure and its associated minimum free energy. This output is in dot-bracket structure and uses the same process as was presented in the Nussinov algorithm.

As mentioned, the MFE table (or score matrix) is an important predictor of the behavior of the Zuker algorithm. Our code contains five different versions of the score matrix. The first score matrix was found from a paper on different RNA folding methods [4]. Here, $(A,U) = (U,A) = -2$, $(G,C) = (C,G) = -3$, $(G,U) = (U,G) = -1$. The next score matrix was found from an MIT lecture series on using dynamic programming for RNA folding [15]. Here, $(A,U) = (U,A) = -.9$, $(G,C) = (C,G) = -2.9$, $(G,U) = (U,G) = -1.1$. We created 3 other score matrices in our codebase which vary from very

large negative values to very small negative values and we will discuss different results later. Our codebase also allows users to create their own score matrix and test the performance on our dataset using that score matrix.

The Zuker algorithm minimizes the free energy of RNA secondary structures to predict stable configurations. RNA folding is governed by thermodynamic principles, where energetically favorable base pairs (e.g., A-U, G-C) and structural features like loops contribute to the total free energy. Lower free energy corresponds to more stable structures, which are typically observed in biological systems and are likelier RNA structures.

The Zuker algorithm computes the free energy of different configurations, penalizing unfavorable features such as mismatches, bulges, and large unpaired loops. By changing the score matrix, the algorithm attempts to avoid forming energetically costly structures, resulting in realistic predictions of compact and stable secondary structures. This contrasts with the Nussinov algorithm, which often produces large rings or unrealistic patterns due to its simpler scoring system. This is described Durbin chapter 10.2 [7] as well as a springer paper [4].

The time complexity is $O(n^3)$ due to nested loops for evaluating possible pairings and substructures, and the space complexity is $O(n^2)$ for storing the dynamic programming table [7]. The results and performance of this algorithm will be described in the next section.

2.5 Four Russians Nussinov Optimization

The Four Russians method is an algorithmic optimization to the Nussinov algorithm that precomputes blocks of solutions to speed up the population of the DP table.

The Four Russians optimization splits the dynamic programming matrix into smaller blocks and precomputes the results for all possible configurations within these blocks. By substituting multiple small calculations with a single lookup in a precomputed table, this method reduces the number of operations performed during runtime [5].

For the Nussinov algorithm, this involves:

- Dividing the RNA sequence into smaller segments.
- Precomputing all possible substructures for each block and storing these in a lookup table.
- Using the precomputed results to fill in the dynamic programming table for larger subsequences.

We now explain implementation details.

- **Matrix Partitioning:** The DP matrix is divided into blocks of size $b \times b$. For a sequence of length n, this results in $(n/b) \times (n/b)$ blocks.
- **Precomputation:** All possible configurations of base pairs within a block are precomputed. This requires 2^{b^2} evaluations for a block of size b, which is feasible for small b (e.g., $b = 4$).
- **Table Lookup:** During the computation of the DP matrix, instead of recalculating the base pairing and split decisions within each block, the algorithm retrieves the precomputed results.
- **Complexity Reduction:** The original Nussinov algorithm has a time complexity of $O(n^3)$. By employing the Four Russians technique, the complexity is reduced to $O(n^3/\log(n))$, where the logarithmic term accounts for the block size.

It is also worth noting that this solution is also optimal. The only changes to the result are in how the DP table is computed. In our version, we also changed how ties were broken during traceback, hence why Nussinov gives different optimal structures when run with the Four-Russians optimization. However, the solution is equally fast and optimal regardless of how ties are broken [5]. Pseudocode can be found in section [4.7].

2.6 Convolution Neural Network Approach

We now discuss our approach to the Neural Network we made to predict RNA structures. To start, this problem can essentially be transformed into a 3-class classification problem, where, for each sequence index, we predict a ‘)’, ‘(’ or ‘.’. This, therefore, can be transformed into a problem that predicts that.

Therefore, our approach was such that the neural network folding predicts RNA secondary structures by classifying each nucleotide’s role in the sequence. The model was trained and validated using a dataset of about 500 RNA sequences and their known secondary structures. Each RNA sequence was encoded using one-hot encoding, which transforms nucleotides (A, C, G, U) into binary vectors, e.g., A = [1, 0, 0, 0]. This format allows the neural network to process the data numerically. See Figure [1] for an explanation of the data pipeline.

The model architecture is based on a Convolutional Neural Network (CNN), which is effective at identifying spatial patterns within sequential data. The input layer accepts the one-hot encoded sequences with a fixed maximum length. The network

includes three Conv1D layers, which apply filters to extract features from the input sequence, such as pairing potential or structural motifs. ReLU activation functions are used to introduce non-linearity, enabling the network to learn complex relationships. See [1] for a diagram of this architecture.

To prevent overfitting, dropout layers are added after the Conv1D layers. These layers randomly deactivate a fraction of neurons during training, ensuring the network generalizes well to unseen data. The final layer is a TimeDistributed Dense layer with softmax activation, which predicts the secondary structure label ('.', '(', or ')') for each position in the sequence independently. The output is a sequence of predictions, aligning with the input sequence length. Figure [1] presents a schematic diagram of this architecture as described to refer to.

2.6.1 Training and Evaluation

The network was trained using categorical cross-entropy as the loss function, which is suitable for multi-class classification problems. The optimizer used was Adam, which efficiently adjusts learning rates during training. The model's performance was evaluated using the Levenshtein distance, which measures the similarity between the predicted and ground-truth secondary structures. By learning directly from labeled RNA data, the model can infer structural roles for nucleotides without explicitly solving energy minimization equations like traditional folding algorithms.

The network was trained using categorical cross-entropy as the loss function, which is suitable for multi-class classification problems. The optimizer used was Adam, which efficiently adjusts learning rates during training. The model's performance was evaluated using the Levenshtein distance, which measures the similarity between the predicted and ground-truth secondary structures. By learning directly from labeled RNA data, the model can infer structural roles for nucleotides without explicitly solving energy minimization equations like traditional folding algorithms.

Our results test on our test data, after we obeyed an 80-20 train-test split. This ensures we are not testing accuracy on data our CNN has already seen, so where our results compare other algorithms on all 500+ sequences, ours is on a smaller test set. This is why the CNN results are collected separately.

2.7 Levenshtein Distance

To evaluate our algorithms, we used Levenshtein distance. Each of our algorithms produces a dot-bracket approximation of a RNA secondary structure. We also have the correct secondary structure in dot-bracket notation from our data set. The Levenshtein distance gives us a way to evaluate how far our algorithm's approximation is from the correct secondary structure. Specifically, it measures how far two strings are in terms of substitutions, insertions, and deletions [16].

For each substitution, insertion, and deletion, 1 is added to the final score. For example, from [17], take the two strings "kitten" and "sitting." For the first letter in each string, 'k' and 's', a substitution is required, 's' for 'k'. So our current score is 1. The next letters, "itt" are the same so no changes are required. Then, for the next letter, 'i' and 'e', another substitution is required, 'i' for 'e'. So, our current score increases by 1 to become 2. From there, sitting is one letter longer than "sittin." So, we need an insertion of a 'g' at the end, resulting in a final score of 3 for "kitten" and "sitting." Also, the closer two strings are to each other, the lower the final score is, as there would be less substitutions, insertions, and deletions.

Additionally, with the Levenshtein distance, there are upper and lower bounds to note. For a lower bound, the distance is greater than or equal to the differences of the lengths of the strings, as in order to make the strings the same length, insertions are required. As an upper bound, the distance is less than or equal to the length of the longest string. Furthermore, the distance is only equal to 0 when the strings are the same [17]. We present the recurrence for this algorithm below:

$$\mathrm{lev}(a,b) = \begin{cases} |b| & \text{if } |a| = 0, \\ |a| & \text{if } |b| = 0, \\ \mathrm{lev}(\mathrm{tail}(a), \mathrm{tail}(b)) & \text{if } \mathrm{head}(a) = \mathrm{head}(b), \\ 1 + \min \begin{cases} \mathrm{lev}(\mathrm{tail}(a), b), \\ \mathrm{lev}(a, \mathrm{tail}(b)), \\ \mathrm{lev}(\mathrm{tail}(a), \mathrm{tail}(b)) \end{cases} & \text{otherwise.} \end{cases}$$

Historically, similar analyses using Levenshtein distances have been done on dot-bracket notations of RNA secondary structures [16]. Specifically,in an analysis of SARS-CoV-2, they used the Levenshtein distance on the dot-bracket representations of "5' and 3' nucleotides of a base pair" [16]. Here, in this paper, the Levenshtein distance gives us the ability to numerically quantify the accuracy of the RNA secondary structure approximations from each algorithm. The pseudocode for our implementation of Levenshtein distance is in section [4.6].

2.8 Data Collection

Over 500 RNA sequences with their 2D folding structures were found at Zenodo [10]. These RNA foldings were in BPSEQ (Base Pair Sequence) form, yet we test our data in dot-bracket form. Thus, we wrote a Python script to transform all 545 predictions to dot-bracket form and export this information as a CSV used for testing. This is the data we used to both train/test our neural network and evaluate performance of the other algorithms.

2.9 Testing Procedure

To validate the RNA secondary structure prediction methods, we created a testing module. This module allows users to test different folding algorithms, compare their outputs, and compute the *Levenshtein distance* between predicted and expected structures. Users can test Zuker, Nussinov, optimized Nussinov, and the neural network-based CNN folding approach. The outputs include predicted structures, distance metrics, and optional CSV logs for analysis.

2.9.1 Key Features and Functionality

The testing module supports multiple modes of operation:

1. **Zuker Score Matrices Comparison**: Test different Zuker algorithm scoring schemes and compute their performance.
2. **Nussinov Tracebacks**: Evaluate predictions from different Nussinov algorithm configurations.
3. **Comprehensive Algorithm Comparison**: Compare all implemented algorithms (Zuker, Nussinov, optimized Nussinov, and CNN) for each RNA sequence.
4. **Custom Zuker Scoring**: Allow users to specify custom scoring matrices to test how different scoring values influence predictions.

2.9.2 User Configuration via Command-Line Flags

Users can define the test type and scoring parameters through command-line arguments:

- `-type`: Specifies the test mode (e.g., `0` for Zuker scoring, `1` for Nussinov, `2` for comprehensive comparison, or `3` for custom Zuker scoring).
- `-custom_scores`: (Optional) Allows users to input three custom scoring values for use in the Zuker algorithm. This is only applicable when `-type` is set to `3`.

Type 0 tests allow the user to test the differences between the five provided Zuker matrices. Type 1 allows the user to compare Nussinov traceback 1 to 2 in terms of accuracy. Type 2 is where the user can compare accuracy of Nussinov, Optimized Nussinov, Zuker, Four Russians, and CNN algorithms. In type 3, the user can create their own Zuker score matrix and see its performance. Note, these comparisons are over our base data of 500 sequences.

For example, running the following command evaluates all algorithms and outputs the results:

```
python testing.py --type 2
```

To use custom scoring with the Zuker algorithm:

```
python testing.py --type 3 --custom_scores -2 -3 -1
```

3 Results

3.1 Evaluation Metrics

Levenshtein distance is the metric used to evaluate accuracy of secondary-structure predictions. It was explained in an earlier section. To measure runtime, we time execution of our code on datasets and measure the results in pure runtime in seconds. We compare Levenshtein distance to the provided secondary structure our dataset provides, as this tells us how true our algorithm's preduction is to the actual secondary structure.

3.2 Qualitative Evaluation

To evaluate the quality of predictions, we can compare the predicted secondary structures from the Nussinov and Zuker algorithms (our fundamental algorithms) against the actual structure using visualizations. algorithms. For example, in figures [5], [6], and [7], we can see clear differences in the predicted secondary structures between the Nussinov and Zuker algorithms compared to the actual structure. Figure [5] shows the actual secondary structure, which contains well-formed loops and stems that appear biologically realistic. The structure is compact and balanced, reflecting stable base pairings and minimal free energy.

In Figure [6], the Nussinov algorithm's prediction shows noticeable inaccuracies. While it captures some stems, it creates larger loops and introduces multiple mispairings. The structure is less compact, with extended regions of unpaired bases, which suggests that the algorithm's focus on maximizing base pairings without considering energy minimization leads to biologically unrealistic results.

Figure [7], showing the Zuker algorithm's prediction, is much closer to the actual structure in Figure [5]. The loops are smaller and more defined, and the overall structure is compact, reflecting the algorithm's consideration of free energy minimization. This results in a more stable and biologically plausible folding pattern. Compared to Nussinov, Zuker's predictions better align with known RNA folding behaviors, demonstrating the advantage of incorporating thermodynamic principles into the algorithm. Finally we analyze the CNN method in Figure [8] and can see that while it is different than the actual prediction, it maintains two larger loops and some long strands of pairings, similar to the actual secondary structure. Clearly, none of these algorithms are completely accurate, but they each display their own unique approximations for folding.

3.3 Synthetic Data Runtime Test

Our first experiment involved generating random RNA sequences of size 100, 200, 400 which we use for runtime comparison. This was synthetically generated with ChatGPT, as we promoted it for RNA sequences of around size 100, 200, 400. We only use this RNA to compare runtime, not correctness, so using AI for this is okay because we don't need access to the real secondary structure. Using this synthetic data, we took the runtime in expectation of 30 runs of our most basic algorithms (Zuker, Nussninov, Optimized Nussinov). We saw Zuker consistently runs marginally faster than the others, and the Optimized Nussinov offers little runtime advantage except for when the sequence has length 400. This can be found in Figure [9]. This experiment ended up not being representative of the other runtime experiments for this algorithm. This could be because we just focused on testing one sequence of each length, making this less accurate as distinct sequences should be considered. Thus, synthetic data was not ideal for testing runtime.

3.4 Runtime on Base Dataset

We now compare the runtime of our four algorithms on our dataset of 500 sequences. We don't compare runtime with our CNN solution because training this CNN takes a few minutes, so we just compare the non-deep-learning algorithms. Refer to Figure [2]. We plot the runtime for each algorithm based on the length of the sequence. Up until sequences of length approximately 125, the algorithms are rather similar in runtime. For sequences over this threshold, the Nussinov and Optimized Nussinov algorithms are around half a second slower than the Zuker and Four Russians methods. The Four Russians Nussinov is the fastest algorithm, and its improvement can get even more noticeable on larger sequences. This proves our Nussinov optimization only catered to that singular synthetic dataset, and when trying it on different sequences, its performance is no better than the standard Nussinov algorithm.

3.5 Levenshtein Distance Between Algorithms

First, Table [1] compares the average Levenshtein distance for each algorithm on this dataset of over 500 sequences. For our CNN algorithm, because we trained on almost 500 sequences, we only find the Levenshtein distance on the validation and testing sets so as to not test on data our CNN has already been trained on. We can see the Zuker and CNN methods give the lowest distances with the Nussinov algorithm performing the worst.

We then aim to find the mean square error (MSE) for each algorithm, defined as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

where y_i represents the observed values, $\hat{y}_i$ are the predicted values, and n is the number of data points. This metric is crucial because a high error indicates that an algorithm can display a wide range of Levenshtein distances between different sequences, reflecting inconsistency. Conversely, if the MSE is low, the algorithm is more reliable, and its accuracy can be predicted fairly well based on sequence length [18]. We connected points with a line hoping to find a clean relationship like Figure [2] displays which is a steady curve. Yet our curve displays a lot of error (more for some algorithms than others) with an overall linear trend. Thus we decided to fit a linear regressor to our data.

We use a Scikit-learn linear regression model to fit a regressor to all of our data and plot this regression in Figure [4]. This shows each algorithm's Levenshtein distance vs sequence length can be approximated by a line fairly intuitively. We then find the mean square error summed over all points. For each dataset of distance vs length, we take the difference of our model's prediction at a specific length and the actual distance at a specific length, square this quantity, and sum over all points. Finally, we normalize by the number of points collected in each dataset. This is called mean square error and is very common for regression. We end with the mean square error column in Table [1]. From this table, we see that the (optimized) Nussinov

algorithm has the highest error, which makes sense because the orange line (with the blue line behind it) display the noisiest curves in Figure [4]. The method with the least error was the Four Russians algorithm (27.04), as the green line in Figure [4] also appears not very noisy.

From these results the Nussinov algorithm clearly has a high Levenshtein distance and accuracy is very noisy, so it varies a lot between sequences. On the other hand, the CNN and Zuker algorithm overall are relatively accurate (with low error as well), and the Four Russians method seems to have medium Levenshtein distance but low accuracy error.

3.6 Bias for certain bases

On all 545 strands, we evaluate what percentage of predicted pairings in the secondary structure are A-U, G-C, G-U in Table [2]. We compare this between algorithms. The Nussinov and Four Russians algorithms produce G-U pairings least frequently, and G-C most frequently. The Zuker algorithm rarely produces G-U pairings, only 4% of the time, and G-C is overwhelmingly 58%. This could be a result of the score matrix used, where G-U pairings had the lowest score so appeared far less frequently. Interestingly, the CNN method is most even between base pairings. A-U is 26%, G-C 42%, and G-U 32%. While these splits are not perfectly even, they are the most equal, so the CNN displays little bias in which bases it will pair. This could be because it does not go based off inherent biological motivation: it just recognizes patterns in data.

3.7 Zuker performance with different score matrices

Table [3] shows how variations in the Zuker algorithm's score matrix affect the base-pairing percentages in predicted secondary structures. Zuker 1 maintains a high A-U pairing percentage (36.98%) with a relatively low G-U percentage (4.1%), reflecting a balanced distribution. In contrast, Zuker 5 strongly favors G-C pairings (65.68%) at the expense of A-U pairings (22.21%), showing that the scoring matrix can significantly bias the algorithm towards specific base-pair types. Zuker 2 and Zuker 3 show moderate G-U percentages (10.82% and 15.32%, respectively), suggesting that these score matrices lead to less restrictive predictions. These results highlight how scoring parameters influence both biological relevance and pairing distributions. This is all dependent on how the Zuker score matrix is set.

We also compare Levenshtein distance of our score matrices in Table [7]. The Levenshtein distances do not vary a ton, only ranging from 21.42 to 22.92. Interestingly, score matrices 1, 3, 4 produce the lowest distance. Zuker 1 is our baseline from the Springer paper [4], so it is a reliable score system.

3.8 Average loop length

Table [4] compares the average loop length across all algorithms. The Zuker algorithm has the shortest average loop length (1.58), showing how it adjusts to thermodynamic stability and compact, biologically plausible structures. The Nussinov algorithms (both standard and optimized) have the longest loop length (3.7), indicating a tendency to create larger unpaired regions. The CNN model's average loop length (3.64) is closer to Nussinov, suggesting it struggles with achieving compactness. The Four Russians Nussinov optimization reduces the loop length to 2.78, balancing compactness.

3.9 Average number pairings

Table [5] compares the average number of base pairings for each algorithm. The CNN model predicts the highest number of base pairings (27.85), likely due to its ability to learn pairing patterns directly from the data with our architecture of the model. The Four Russians Nussinov algorithm also predicts a relatively high number of base pairings (23.46), outperforming both the Zuker algorithm (21.2) and the standard Nussinov algorithm (16.6). This shows that the Four Russians optimization introduces improvements over the basic Nussinov algorithm while balancing computational performance. Zuker's lower pairing count aligns with its emphasis on energy minimization rather than maximizing pairings.

4 Discussion

The results show differences between all four algorithms. The Nussinov algorithm and its optimized version showed similar performance, with a total average Levenshtein distance of 27.61 for the RNA sequences in our dataset. The optimized version slightly improved runtime for longer sequences but did not offer any accuracy advantage over the original. Both algorithms produced larger average loop lengths (3.7) and fewer base pairings (16.6) which indicates lower biological accuracy compared to the other methods that we used. Runtime differences between these two algorithms were seen as the sequence size increased (as shown in Figure [2]).

For the Zuker algorithm, we saw much better accuracy, with an average Levenshtein distance of 18.99 and shorter loop lengths (1.58). This algorithm also had higher base pairings (21.2). Zuker works by minimizing the free energy in a system, which leads to this better performance that we see. Zuker and Nussinov performed similarly in runtime across all sequence lengths,

although there is a slight improvement for Zuker. This makes Zuker a strong choice for applications with accuracy and also computational efficiency.

We also used the method of Four Russians to implement the Nussinov algorithm in a faster way. This showed improvements in runtime as well as the actual accuracy, achieving an average Levenshtein distance of 23.17 across the dataset.

The CNN model had an average Levenshtein distance of 23.2 across the dataset, which is the third best performing model. It produced the highest number of base pairings (about 27.85 average), but also generated large loop sizes (3.64). What we did here was not optimize on the number of base pairings or minimize free energy in the system, we simply trained on expected sequence outputs, which is part of the reason these loops exist. It doesn't fully learn the biological behavior of these predictions, but it serves as a very promising way to show how useful CNNs and neural networks can be for researchers in this field.

We are impressed with how our more unique algorithms (such as Four Russians and CNN) compared to standard baselines over our immense testing. We were able to compare the structural differences of the outputs, their biases in different base pairings, the speed of each algorithm, their accuracy to real folding, and how these values change across differing numbers of sequences with different lengths. Most of all, our novel CNN approach reached accuracy levels of the Zuker and Four Russians algorithms, proving itself a reliable method.

These results were promising, yet strictly going down this neural network path may invite some limitations of our work. For one, we aimed to find an algorithm which reduces computational resources and runtime complexity. This is why we experimented with the Four Russians optimization. Yet, our CNN approach—while may have room for great performance—can be computationally intensive. Training large multi-layer CNNs on large sequences takes time and computational resources. This becomes especially more difficult if we continue to train on more data or tune more parameters. So a CNN approach could lead to high accuracy, but this would be at the expense of training time and possibly using GPUs or more advanced machines. That being said, once a model is trained, prediction is not a very intense process. Still, this is a limitation worth noting.

This comprehensive analysis of the properties across these algorithms yields valuable insight into runtime, accuracy, and seeing generally how each algorithm makes its choices. For example, we gain a sense of how a CNN performs by just recognizing sequences in data. In the larger context of secondary structure prediction as a whole, we are proud to have explored two distinct paths of implementation and optimization: standard algorithms and machine learning. Machine learning is gaining in popularity in the world of secondary structure prediction, and our findings further validate their validity, as it was a highly accurate method.

Further research endeavors could involve training more machine learning systems on more data. This would take massive amounts of compute, but as explored [8, 9], RNA secondary structure prediction has become a field leveraging intense computation. These papers mention RNA folding algorithms utilizing parallel and GPU computing. Our current approaches didn't require these resources, but we could explore how algorithms would perform in these high-compute environments with loads of data. Overall, this project posed the duality of secondary structure prediction algorithms, balancing rudimentary DP problems to ones with more a biological motivation, and utilizing a pure machine learning approach.

References

1. Zemora G, Waldsich C, 2010. Rna folding in living cells. *RNA Biology*, 7(6):634–641.

2. Eddy S, 2004. Folding the code. *Nature Biotechnology*, 22(11):1457.

3. Dimitrieva S, Bucher P, 2012. Practicality and time complexity of a sparsified rna folding algorithm. *Journal of Bioinformatics and Computational Biology*, 10(2):1241007.

4. Fröhlich H, et al., 2012. Rna structure prediction. In *Algorithms in Bioinformatics: Advanced Computational Approaches*, pages 241–258. Springer.

5. Venkatachala B, et al., 2013. Faster algorithms for rna-folding using the four-russians method. *arXiv preprint*, arXiv:1307.7820.

6. Yu L, 2015. Study of rna secondary structure prediction algorithm. Master's Project, San Jose State University.

7. Durbin R, Eddy SR, Krogh A, Mitchison G, 1998. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press.

8. Backofen R, et al., 2014. Fast rna folding using the four-russians method. *BMC Bioinformatics*, 15(S8):S1.

9. Gu X, et al., 2023. Derna enables pareto optimal rna design. *Journal of Computational Biology*, 30:542–558.

10. Marek J, Antczak M, Szachniuk M, 2023. Datasets for benchmarking rna 2d structure prediction algorithms. Zenodo. 0.1.

11. Zhao C, et al., 2017. Cache and energy efficient algorithms for nussinov's rna folding. *BMC Bioinformatics*, 18(1917):1917.

12. Schneiderbauer S, 2008. Rna secondary structure prediction. Master's thesis, Osnabrück University.

13. Varani G, McClain WH, 2000. The g x u wobble base pair: A fundamental building block of rna structure crucial to rna function in diverse biological systems. *EMBO Reports*, 1(1):18–23.

14. Vienna RNA Package, 2024. Rna structures. https://www.tbi.univie.ac.at/RNA/ViennaRNA/refman/io/rna_structures.html. Accessed: 2024-12-16.

15. Will S, 2024. Rna secondary structure prediction using dynamic programming. MIT 18.417.

16. Ziesel A, Jabbari H, 2024. Unveiling hidden structural patterns in the sars-cov-2 genome: Computational insights and comparative analysis. *PLoS One*, 19(4):e0298164.

17. Hossain A, 2024. Levenshtein distance. *ScienceDirect Topics*, . Accessed: 2024-12-21.

18. Stewart K, 2024. Mean squared error. Accessed 21 December 2024.

19. Jeffa P, 2024. A zuker-like rna folding algorithm. https://www.kaggle.com/code/pyjeffa/a-zuker-like-rna-folding-algorithm. Accessed: 2024-12-20.

20. Consortium R, 2024. Rnacentral r2dt: A tool for rna secondary structure visualization and comparison. https://rnacentral.org/r2dt. Accessed: 2024-12-20.

21. Nam E, 2024. Understanding the levenshtein distance equation for beginners. https://medium.com/@ethannam/understanding-the-levenshtein-distance-equation-for-beginners-c4285a5604f0. Accessed: 2024-12-21.

Figures

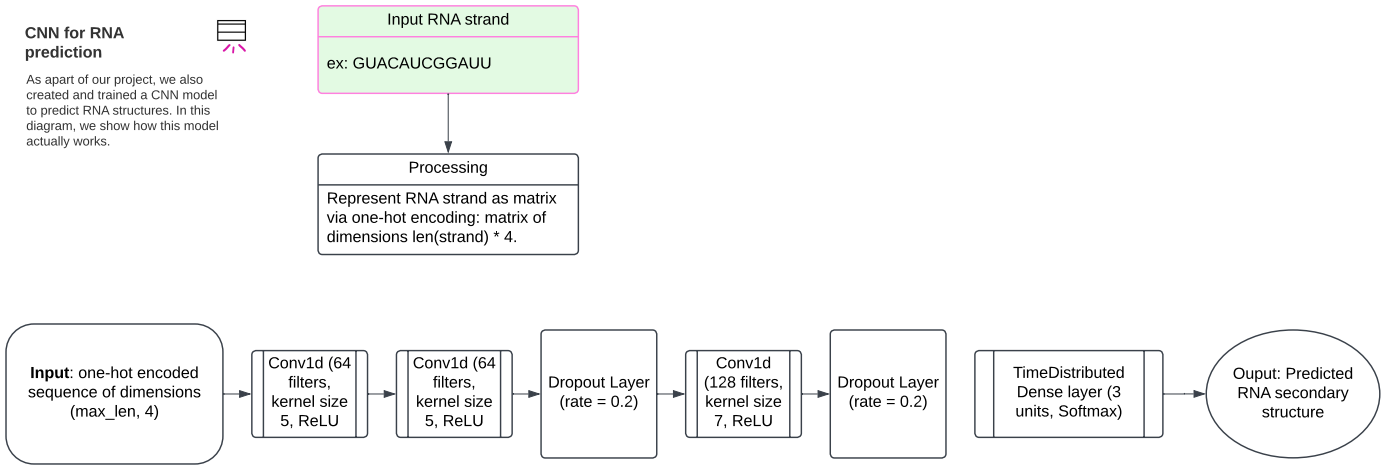

Figure 1. Schematic diagram of the CNN architecture used for RNA secondary structure prediction. This diagram offers an explanation for how our CNN was constructed to perform RNA secondary structure prediction. First, we demonstrate our data pre-processing, where RNA molecules get represented as a one-hot encoding ([1,0,0,0] for A, etc.) which becomes the input to our neural network. Note, we also pad this matrix with blank matrices as the maximum length for a sequence is 390. We have 3 Conv1d ReLU layers with two dropout layers. Then there is a TimeDistributed Dense layer with a Softmax activation before our output: a predicted RNA secondary structure.

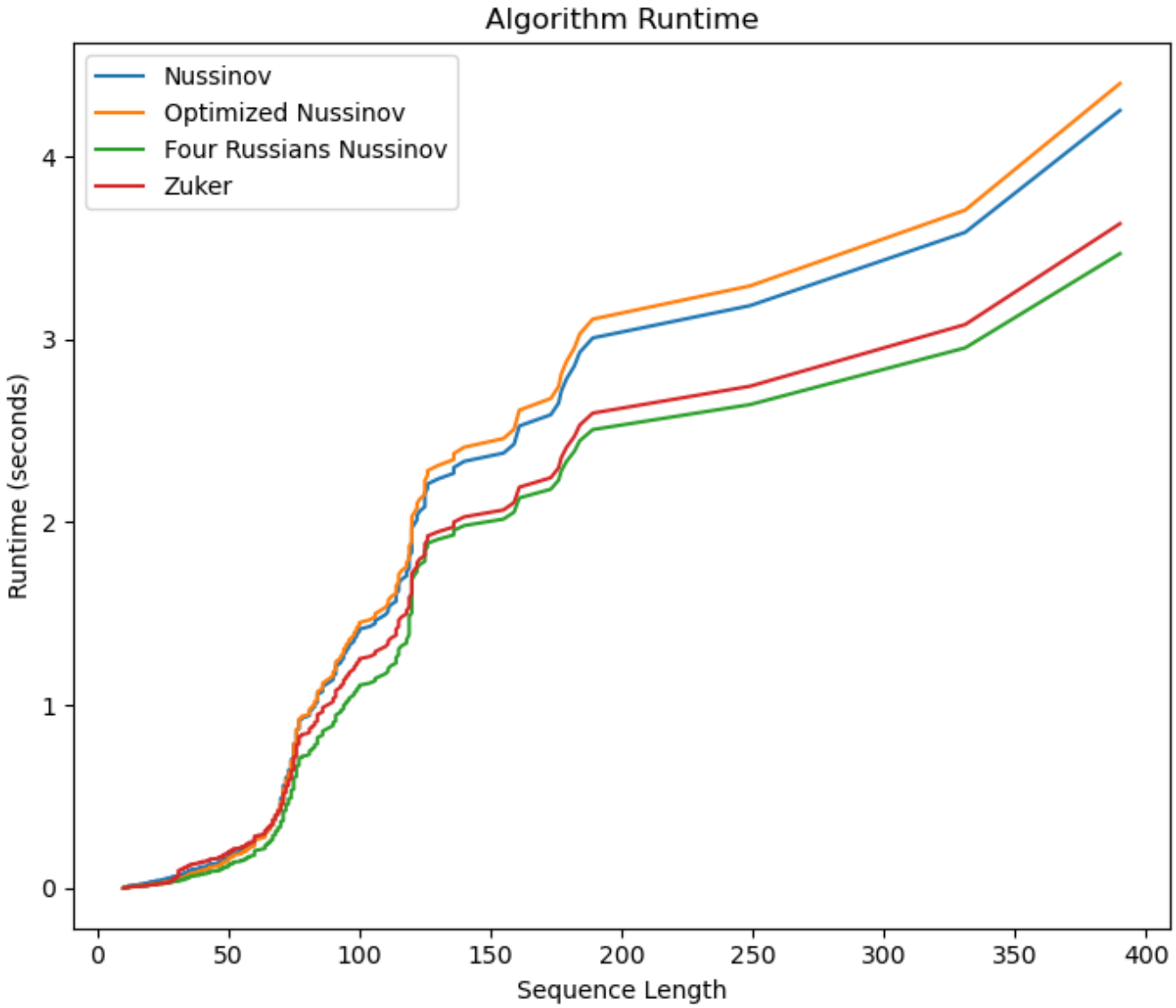

Figure 2. Runtime vs Sequence Length. This figure compares the Nussinov, Optimized Nussinov, Four Russians Nussnov, and Zuker algorithms by runtime for sequences of length 10 to 400. Clearly, as sequences increase in length, runtime increases. Here, though, we see certain regions display larger changes than others. The Zuker and Four Russians methods seem to be noticably faster than the others, which are relatively similar in behavior. The CNN approach was not considered here as it was not created for runtime purposes (training takes a long time).

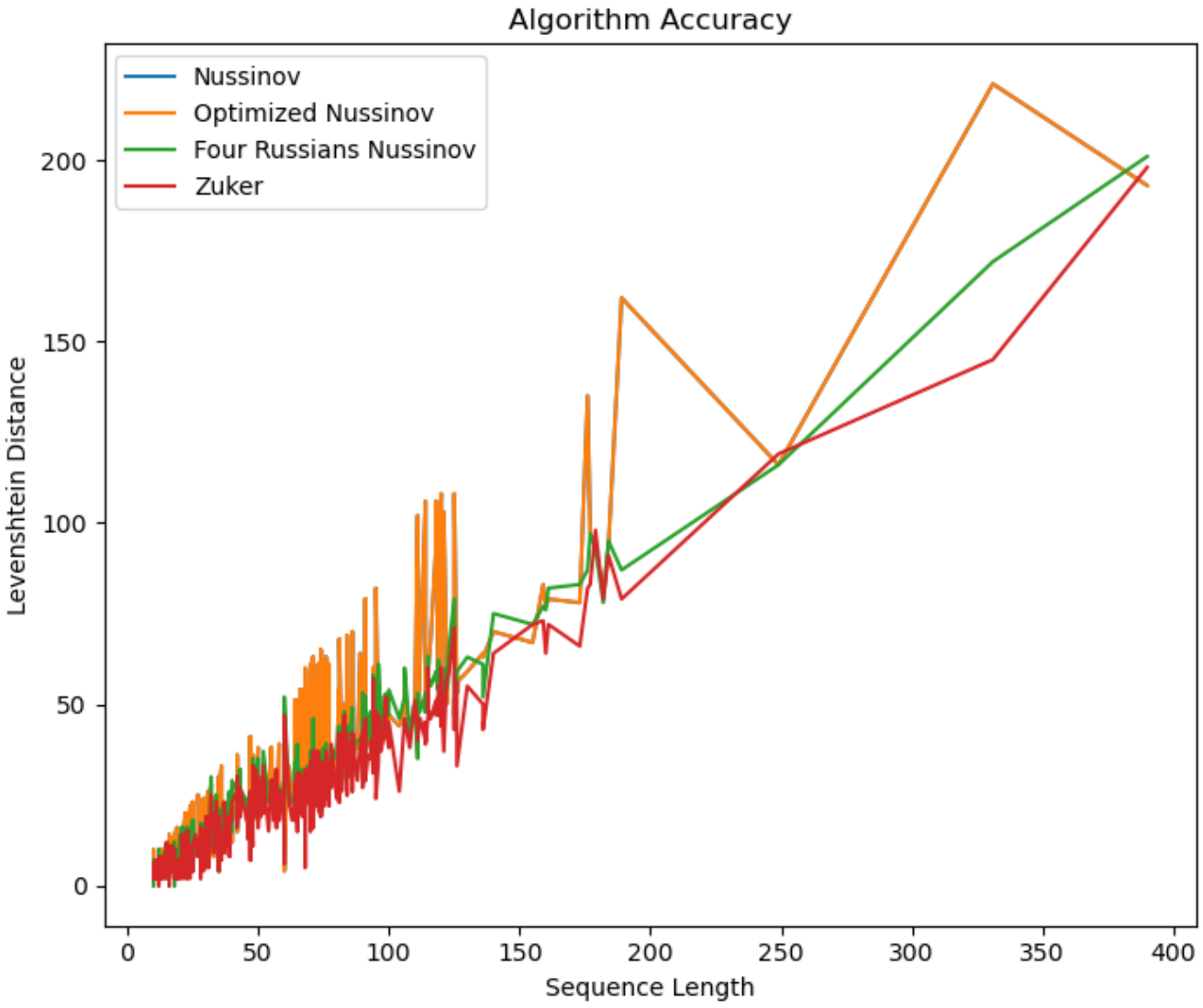

Figure 3. Accuracy vs Sequence Length. This figure compares the Nussinov, Optimized Nussinov, Four Russians Nussinov, and Zuker algorithms by Levenshtein for sequences of length 10 to 400. As sequences increase in length, the Levenshtein distance to the true folding seems to increase linearly. Line segments connect all of these points, but this data is very noisy, so it is more useful to fit a regression to this data and measure mean square error for each algorithm. From pure observations, the (Optimized) Nussinov algorithm is the noisiest, and the Four Russians method is the least noisy. We consider accuracy of CNN separately in tables.

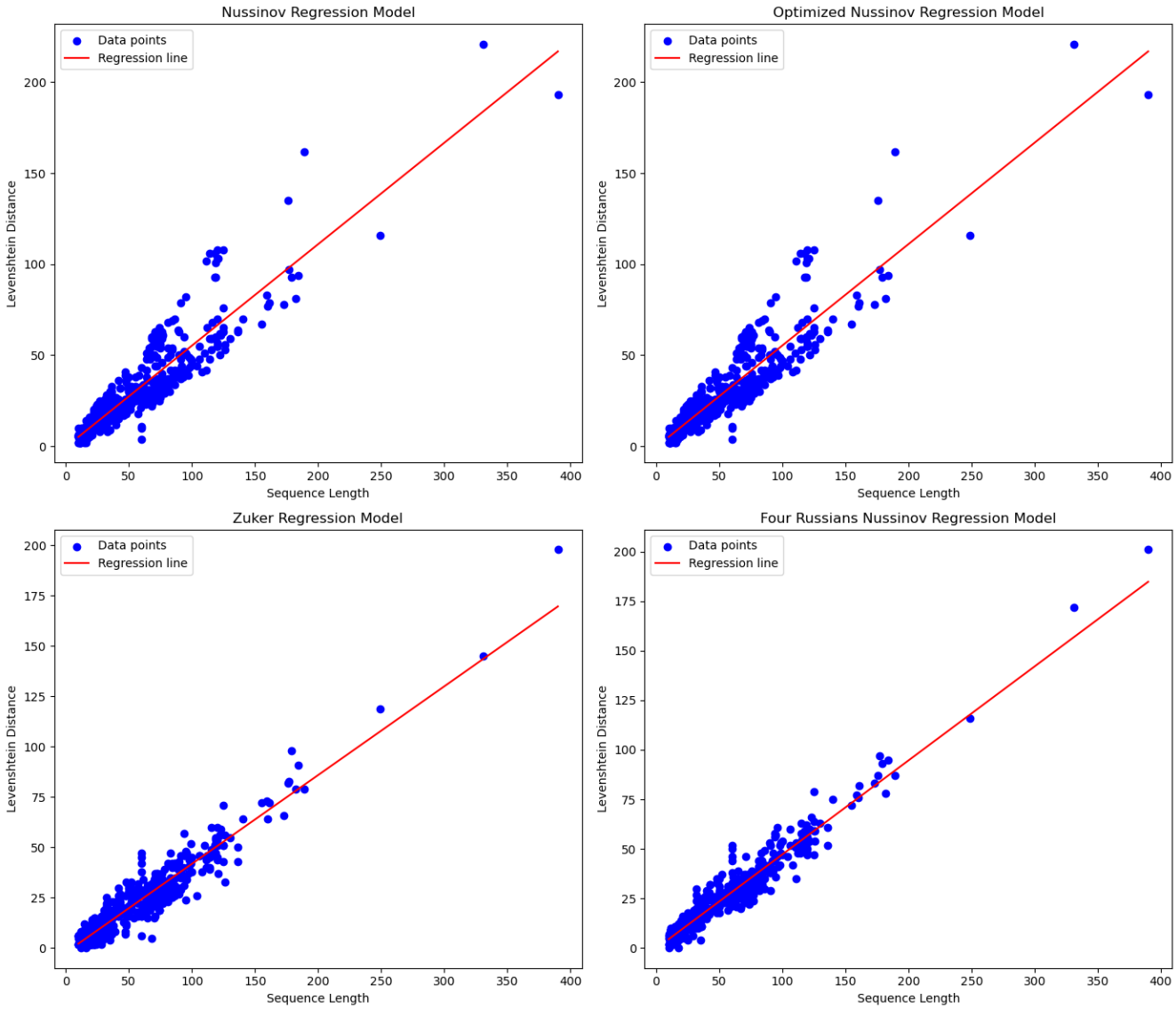

Figure 4. Regression Models of Levenshtein Distance. These are the linear regression models fit to the data plots of Levenshtein distance vs sequence length. These plots are from figure 2. Low mean square error means the Levenshtein distance is more predictable based off of the sequence length but high distance means the algorithm varies more from the true secondary structure. Linear regressors represent this data fairly well, and we can analyze mean square error to understand how noisy specific algorithms are.

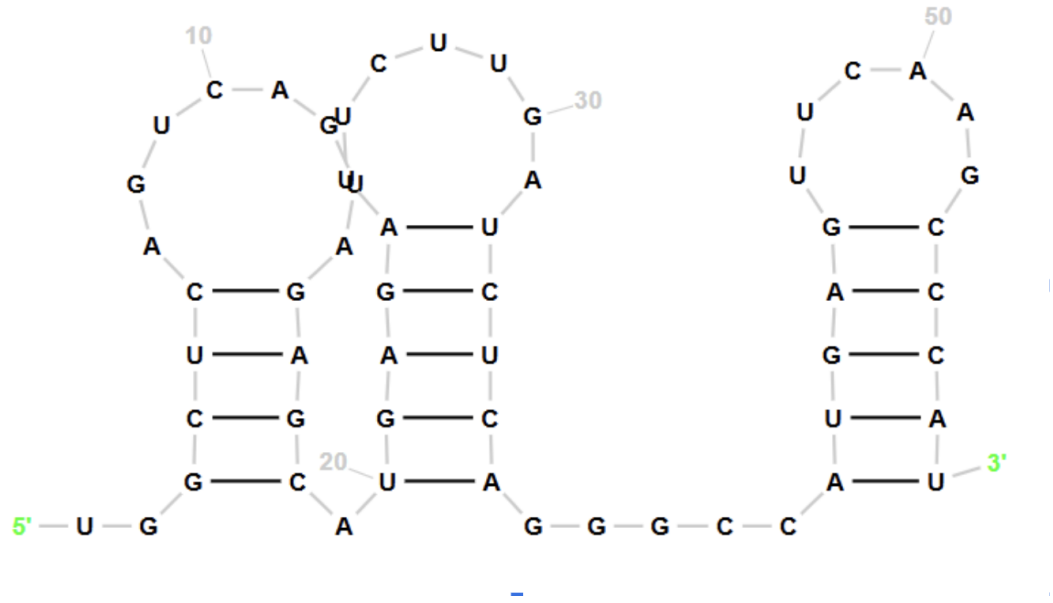

Figure 5. Actual secondary structure of cat RNA with RNACentral visualization. For visualization purposes, we first obtained an example cat RNA sequence with its secondary structure in dot-bracket form. This was found from an online Kaggle dataset [19]. We then used the RNACentral visualization tool which takes in an RNA sequence and dot-bracket structure to create a visual representation of this secondary structure [20].

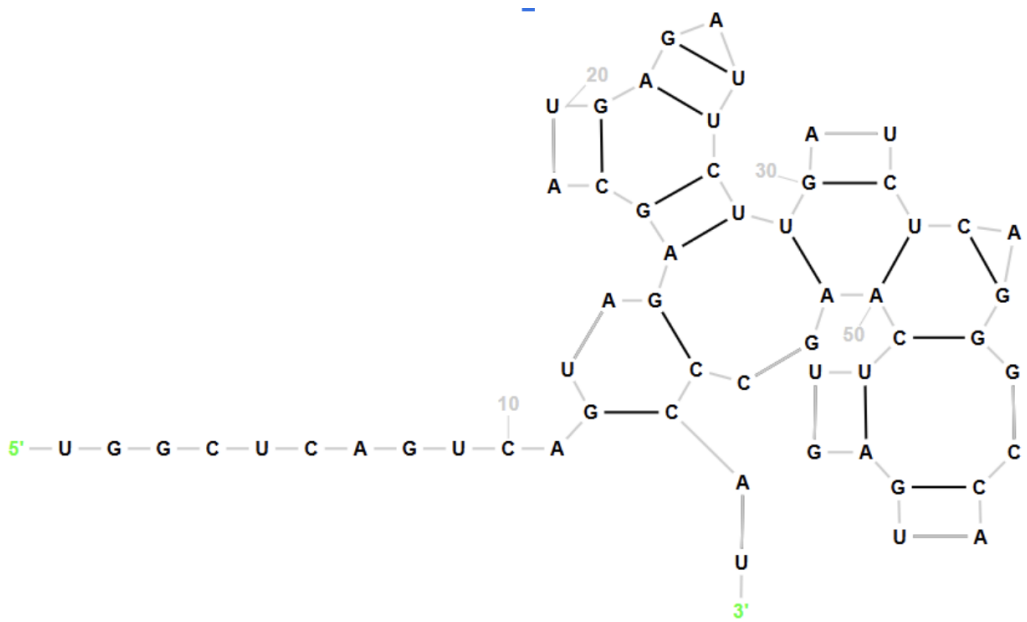

Figure 6. Nussinov prediction of cat RNA with RNACentral visualization. For visualization purposes, we first obtained an example cat RNA sequence with its secondary structure in dot-bracket form. This was found from an online Kaggle dataset [19]. We found the Nussinov prediction dot-bracket form using our Nussinov algorithm and used the RNACentral visualization tool to create a visual representation of this prediction [20].

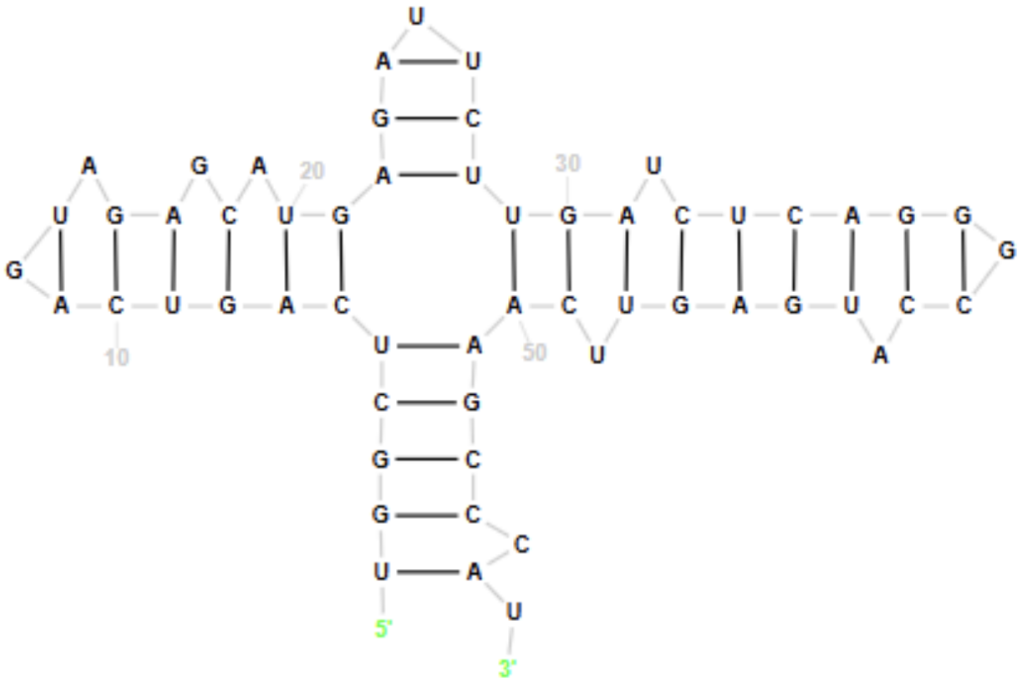

Figure 7. Zuker prediction of cat RNA with RNACentral visualization. For visualization purposes, we first obtained an example cat RNA sequence with its secondary structure in dot-bracket form. This was found from an online Kaggle dataset [19]. We found the Zuker prediction dot-bracket form using our Zuker algorithm and used the RNACentral visualization tool to create a visual representation of this prediction [20].

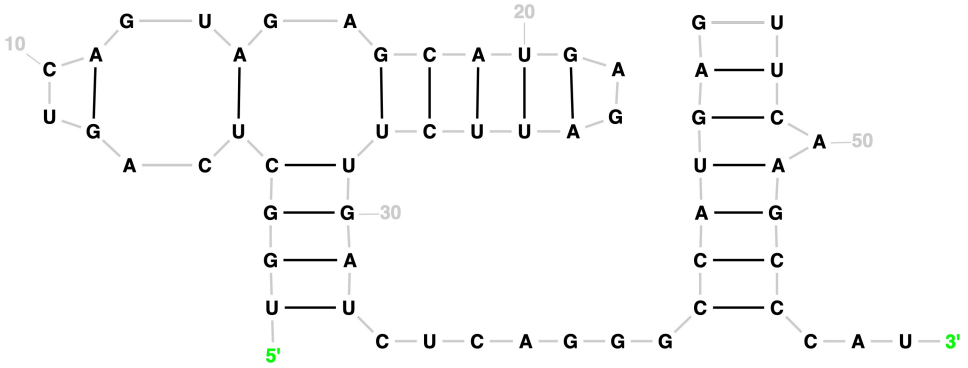

Figure 8. CNN prediction of cat RNA with RNACentral visualization. For visualization purposes, we first obtained an example cat RNA sequence with its secondary structure in dot-bracket form. This was found from an online Kaggle dataset [19]. We found the CNN prediction dot-bracket form using our CNN model algorithm and used the RNACentral visualization tool to create a visual representation of this prediction [20].

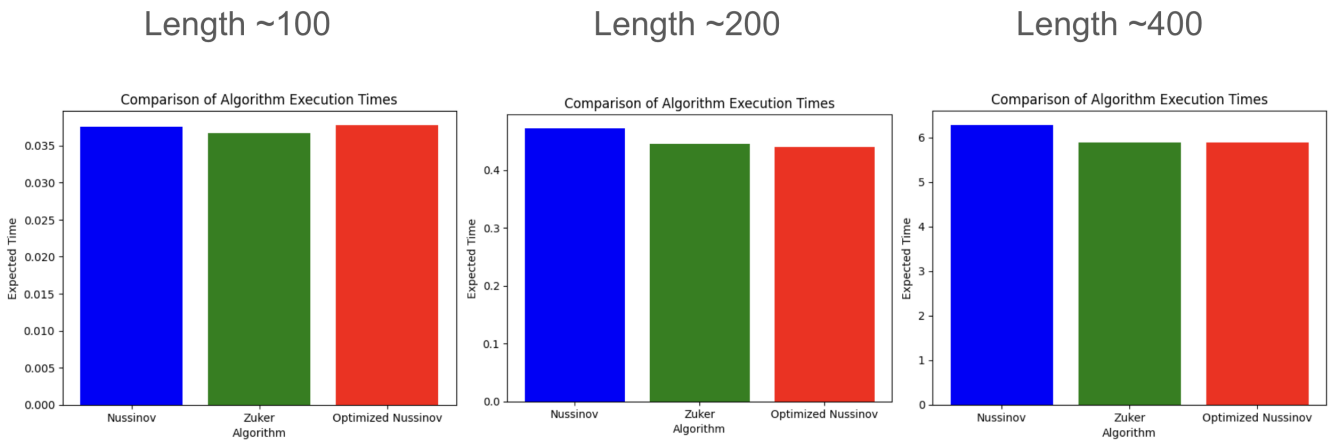

Figure 9. Runtime per algorithm (30 runs). This is a primitive figure where we collected average runtime in expectation over 30 runs in a synthetic dataset generated from ChatGPT of length 100, 200, 400. We used Python to time how long each algorithm took and plotted the results. Zuker is consistently the fastest, but we see the Nussinov optimization sometimes performs better. Based on other results, we can see flaws in this experiment as over more sequences of varying length, this algorithm ends up rarely being faster.

		k	i	t	t	e	n
	0	1	2	3	4	5	6
s	1	1	2	3	4	5	6
i	2	2	1	2	3	4	5
t	3	3	2	1	2	3	4
t	4	4	3	2	1	2	3
i	5	5	4	3	2	2	3
n	6	6	5	4	3	3	2
g	7	7	6	5	4	4	3

Figure 10. Levenshtein distance example. This is a diagram demonstrating how the Levenshtein distance is calculated [21]. Specifically, the two strings are kitten and sitting. From, the Levenshtein distance description above, the final distance between these strings is 3 after two substitutions and one insertion. [17].

Tables

Table 1. Levenshtein Distance Comparison. First, this table shows the average Levenshtein distance when run on the dataset of 500 sequences. This table represents the mean square error found for each of the key algorithms after fitting a linear regression to the data plots of Levenshtein distance vs sequence length. These plots are from Figure [3]. Low mean square error means the Levenshtein distance is mire predictable based off of the sequence length but high distance means the algorithm varies more from the true secondary structure.

Algorithm	Average Levenshtein Distance	Mean Square Error
Nussinov Algorithm	27.61	111.499
Optimized Nussinov Algorithm	27.61	111.499
Zuker Algorithm	18.99	35.421
Four Russians Nussinov Algorithm	23.17	27.036
CNN Algorithm (on test/validation set of $\tilde{1}00$ sequences)	23.21	N/A

Table 2. Bias of Bases. This table represents the percentages of base pairings found in each secondary structure. This is used to see if an algorithm tends to favor pairing one base over the other.

Algorithm	A-U Percentage	G-C Percentage	G-U Percentage
Nussinov Algorithm	33.94	52.7	13.35
Optimized Nussinov Algorithm	33.94	52.7	13.35
Zuker Algorithm	36.98	58.95	4.08
Four Russians Nussinov Algorithm	34.8	52.51	12.69
CNN	25.5	42.44	32.06

Table 3. Zuker Score Matrix Comparison. This table represents the percentages of base pairings found in each secondary structure when varying the Zuker score matrix. This is used to see if a score matrix tends to favor pairing one base over the other. We also plot the average Levenshtein distance of each score matrix over our dataset of 500 sequences.

Algorithm	A-U Percentage	G-C Percentage	G-U Percentage	Average Levenshtein Distance
Zuker 1	36.98	58.95	4.1	21.58
Zuker 2	26.68	62.49	10.82	22.10
Zuker 3	33.65	51.02	15.32	21.42
Zuker 4	33.65	51.02	15.32	21.42
Zuker 5	22.21	65.68	12.11	22.92

Table 4. Average Loop Length. This table represents the average loop length on the dataset of 500 sequences compared between all algorithms. Loop length is determined by number of RNA positions.

Algorithm	Average Loop Length
Nussinov Algorithm	3.7
Optimized Nussinov Algorithm	3.7
Zuker Algorithm	1.58
Four Russians Nussinov Algorithm	2.78
CNN	3.64

Table 5. Average Number of Base Pairings. This table represents the average number of base pairings on the dataset of 500 sequences compared between all algorithms.

Algorithm	Average Number of Base Pairings
Nussinov Algorithm	16.6
Optimized Nussinov Algorithm	16.6
Zuker Algorithm	21.2
Four Russians Nussinov Algorithm	23.46
CNN	27.85

Algorithms

4.1 The Nussinov Algorithm for RNA Secondary Structure Prediction

Input: RNA sequence $S = s_1, s_2, \ldots, s_n$ of length n
Output: Maximum number of base pairs in the secondary structure

---

```
// Initialize DP table
Initialize DP[i][j] ← 0 for all i, j such that j ≤ i + 1
// Fill DP table for increasing subsequence lengths
for l ← 2 to n do
    for i ← 1 to n − l + 1 do
        j ← i + l − 1
        DP[i][j] ← max(DP[i+1][j], DP[i][j−1], DP[i+1][j−1] + pair(S[i], S[j]))
        for k ← i to j − 1 do
            DP[i][j] ← max(DP[i][j], DP[i][k] + DP[k+1][j])
        end
    end
end
// Backtrack to reconstruct the structure (optional)
Backtrack through the DP table to recover base pairs
return DP[1][n]                                   // Maximum number of base pairs
Function pair(a, b):
    if (a, b) is a valid base pair (A-U, G-C, G-U) then
        return 1
    else
        return 0
    end
```

4.2 Nussinov Traceback 1

The first valid Nussinov traceback using DP and $pair$ from the Nussinov algorithm [4.1].

```
// Stack-based traceback to recover base pairs
Function Backtrack(i, j):
    if i ≥ j then
        return                                          // Stop condition: no base pairs possible
    else if DP[i+1][j] == DP[i][j] then
        Backtrack(i+1, j)                                                  // Base i is unpaired
    else if DP[i][j-1] == DP[i][j] then
        Backtrack(i, j-1)                                                  // Base j is unpaired
    else if DP[i+1][j-1] + pair(S[i], S[j]) == DP[i][j] then
        Record pair (i, j)
        Backtrack(i+1, j-1)                                          // Bases i and j are paired
    else
        for k ← i to j-1 do
            if DP[i][j] == DP[i][k] + DP[k+1][j] then
                Backtrack(i, k)
                Backtrack(k+1, j)
                return
            end
        end
    end
Backtrack(1, n)
return Set of base pairs and DP[1][n]
```

4.3 Nussinov Traceback 2

The second valid Nussinov traceback using DP and $pair$ from the Nussinov algorithm [4.1].

```
// Stack-based traceback to recover base pairs
Function Backtrack(i, j):
    if i ≥ j then
        return                                          // Stop condition: no base pairs possible
    else if DP[i+1][j] == DP[i][j] then
        Backtrack(i+1, j)                                                  // Base i is unpaired
    else
        for k ← i to j-1 and pair(S[i], S[j]) do
            if DP[i][j] == DP[i][k-1] + DP[k+1][j-1] + 1 then
                Record pair (i, j)
                Backtrack(i, k-1)
                Backtrack(k+1, j-1)
                return
            end
        end
    end
Backtrack(1, n)
return Set of base pairs and DP[1][n]
```

4.4 Optimized Nussinov Algorithm

Input: RNA sequence $S = s_1, s_2, \ldots, s_n$ of length n
Output: Maximum number of base pairs in the secondary structure

---

// Initialize DP table
Initialize $DP = \{\}$. $DP[1] = [0] * n$, $DP[2] = [0] * (n-1)$, ..., $DP[n] = [0]$
Initialize $delta\_lookup = \{(A,U) = 1, ..., (C,G) = 1, (A,G) = 0, ..., (G,U) = 0\}$ // Fill DP table for increasing subsequence lengths
for $l \leftarrow 2$ ***to*** n **do**
    for $i \leftarrow 1$ ***to*** $n - l + 1$ **do**
        $j \leftarrow i + l - 1$
        $DP[i][j-(i-1)] \leftarrow \max\left(DP[i+1][j-i],\ DP[i][j-1-(i-1)],\ DP[i+1][j-1-i] + delta\_lookup[(i,j)]\right)$
        for $k \leftarrow i$ ***to*** $j-1$ **do**
            $DP[i][j-(i-1)] \leftarrow \max\left(DP[i][j-(i-1)],\ DP[i][k-(i-1)] + DP[k+1][j-k]\right)$
        end
    end
end
// Backtrack to reconstruct the structure (optional)
Backtrack through the DP table to recover base pairs
return $DP[1][n]$ // Maximum number of base pairs

4.5 Zuker Algorithm

Algorithm 1: Zuker Algorithm for RNA Folding

Input: RNA sequence S of length n
Output: Minimum free energy and optimal secondary structure
Initialize DP arrays:
    `MFE[i][j]` $\leftarrow \infty$ for all $0 \leq i < j \leq n$
    `Traceback[i][j]` $\leftarrow$ None
Base cases:
    `MFE[i][i]` $\leftarrow 0$ for all $1 \leq i \leq n$
    `MFE[i][i+1]` $\leftarrow 0$ (no base pair possible in a single base)
for length = 2 to n **do**
  for $i = 1$ to $n - \text{length} + 1$ **do**
    $j \leftarrow i + \text{length} - 1$
    Case 1: No base pairing
      `MFE[i][j]` $\leftarrow \min($`MFE[i][j]`$,$`MFE[i+1][j]`$)$
    Case 2: Hairpin loop
    if `is_complementary`(S[i], S[j]) **then**
      `hairpin_energy` $\leftarrow$ `calculate_hairpin_energy`(i, j)
      `MFE[i][j]` $\leftarrow \min($`MFE[i][j]`$,$`hairpin_energy`$)$
    end if
    Case 3: Base pair stacking
    if `is_complementary`(S[i], S[j]) **then**
      `MFE[i][j]` $\leftarrow \min($`MFE[i][j]`$,$`MFE[i+1][j-1]` $+$ `stack_energy`$(i, j))$
    end if
    Case 4: Bulge/Internal loop
    for $k = i + 1$ to $j - 1$ **do**
      for $l = k + 1$ to $j - 1$ **do**
        `MFE[i][j]` $\leftarrow \min($`MFE[i][j]`$,$`MFE[i][k]` $+$ `MFE[l][j]` $+$ `bulge_internal_energy`$(i, j, k, l))$
      end for
    end for
    Case 5: Multi-branch loops
    for $k = i + 1$ to $j - 1$ **do**
      `MFE[i][j]` $\leftarrow \min($`MFE[i][j]`$,$`MFE[i][k]` $+$ `MFE[k+1][j]` $+$ `multi_branch_energy`$(i, j, k))$
    end for
  end for
end for
Traceback: Call `traceback(MFE, i=1, j=n)` to reconstruct the secondary structure.
Return: `MFE[1][n]` as the minimum free energy and the optimal secondary structure.

4.6 Levenshtein Distance

Input: string1 and string2
Output: Levenshtein distance score

Input: Two strings, `string1` of length m and `string2` of length n
Output: Levenshtein distance score between `string1` and `string2`

Function *LevenshteinDistance(string1, string2)*:
    `// Initialize a 2D table to store distances`
    Create a matrix DP of size $(m+1) \times (n+1)$
    `// Base cases: initialize first row and column`
    for $i \leftarrow 0$ ***to*** m **do**
        $DP[i][0] \leftarrow i$
    end
    for $j \leftarrow 0$ ***to*** n **do**
        $DP[0][j] \leftarrow j$
    end
    `// Fill the DP table`
    for $i \leftarrow 1$ ***to*** m **do**
        for $j \leftarrow 1$ ***to*** n **do**
            if `string1[i-1] == string2[j-1]` **then**
                $cost \leftarrow 0$
            else
                $cost \leftarrow 1$
            end
            $DP[i][j] \leftarrow \min\left(DP[i-1][j]+1,\ DP[i][j-1]+1,\ DP[i-1][j-1]+cost\right)$
        end
    end
    `// Return the Levenshtein distance`
    return $DP[m][n]$

4.7 Four Russians Nussinov Optimization

Input: RNA sequence $S = s_1, s_2, \ldots, s_n$ of length n, block size b
Output: Optimal secondary structure and maximum number of base pairs

---

// Initialize DP table and precompute configurations
Initialize $DP[i][j] \leftarrow 0$ for all $1 \leq i \leq n,\ j \leq n$
Initialize T to store solutions for all 2^{b^2} possible configurations
for *each RNA sequence R of length b* **do**
    Compute $T[R]$ using the Nussinov recurrence:

$$T[R][i][j] = \max(T[R][i+1][j],\ T[R][i][j-1],\ T[R][i+1][j-1] + \text{pair}(R[i], R[j]),\ \max_{k=i+1}^{j-1}(T[R][i][k] + T[R][k+1][j]))$$

end
// Fill DP table using precomputed blocks
for $l \leftarrow b$ ***to*** n ***step*** b **do**
    for $i \leftarrow 1$ ***to*** $n - l + 1$ ***step*** b **do**
        $j \leftarrow i + l - 1$
        Retrieve precomputed values for block B_{ij} from T
        for *each overlapping region between* B_{ij} *and adjacent blocks* **do**
            Update $DP[i][j]$ as:

$$DP[i][j] = \max(DP[i][j], DP[i][k] + DP[k+1][j])$$

            where k is the boundary index between blocks
        end
    end
end
// Traceback to reconstruct the structure
Backtrack through DP to recover base pairs
return Optimal secondary structure // Maximum number of base pairs