A More Efficient Dynamic Programming Algorithm for Designing a Coding Sequence by Jointly Optimizing Its Structural Stability and Codon Usage

Yan-Ru Ju, Long-Shang Cho, and Chin Lung Lu

Abstract—**Currently, a dynamic programming (DP) algorithm CDSfold has been proposed to design a CDS by minimizing the minimum free energy (MFE) of its secondary structure. However, it has been questioned recently that such a DP algorithm is difficult to be modified to design a CDS when attempting to jointly optimize its secondary structure stability and codon adaptation index (CAI). In this study, we successfully modify the DP algorithm of CDSfold to exactly solve this kind of CDS design problem in $\mathcal{O}(L^3)$ time and $\mathcal{O}(L^2)$ space, where L is the CDS length. We further accelerate this DP algorithm by beam search, enabling it to design a high-quality approximate CDS in $\mathcal{O}(L)$ time, and implement it as the program LinearCDSfold. Our experimental results show that when running with exact search, LinearCDSfold has comparable accuracy to two state-of-the-art CDS design tools LinearDesign and DERNA in terms of both MFE and CAI. In terms of running time, however, LinearCDSfold is slower than LinearDesign, but significantly faster than DERNA, even though they all run in $\mathcal{O}(L^3)$ time and $\mathcal{O}(L^2)$ space. Moreover, LinearCDSfold using beam search can design an approximate CDS in very short time with very high quality in terms of both MFE and CAI.**

Index Terms—**Dynamic programming (DP) algorithm, CDS design, secondary structure, MFE, codon usage, CAI.**

I. Introduction

The rapid development of messenger RNA (mRNA) vaccines during the pandemic period of coronavirus disease 2019 (COVID-19) has demonstrated the enormous potential of mRNA-based therapeutics due to their safety and efficacy [1], [2]. However, an mRNA molecule is unstable and prone to be degraded, leading to insufficient protein expression that weakens the capacity of a vaccine to stimulate strong immune responses. The degradation of an mRNA actually can be mitigated by forming a more stable secondary structure. But, in this situation, it is generally assumed that the difficulty of cellular translation machinery to process the secondary structure in an mRNA will be increased, resulting in that the protein output of the mRNA is lowered. However, some recent studies have suggested that this problem introduced by secondary structure might not exist, by showing that increasing the secondary structure in the coding sequence (CDS) of an mRNA can substantially improve its protein expression [3], [4], [5], [6]. In addition, the preference of codon usage has long been recognized as another important factor in the protein expression, because it can be different in various host genomes and usually has a positive correlation with the level of protein expression [7], [8]. It has been reported that optimizing the codon usage of a CDS barely improves its secondary structure stability, and vice versa [6]. Therefore, it is important and desirable to have an efficient algorithm that can design the CDS by optimizing its secondary structure stability and codon usage together.

Actually, the space of feasible candidates for designing an mRNA is exponential because of the redundancies of the genetic code. Basically, there are $4^3 = 64$ codons for 20 amino acids, implying that an amino acid can be encoded by multiple codons (at most 6 codons per amino acid). Currently, a few algorithms described below have been proposed to design the CDS by optimizing the stability of its secondary structure and/or its codon usage. By extending the Zuker algorithm [9] with considering the amino acid constraints, Cohen and Skiena [10] and Terai et al. [11] independently proposed a dynamic programming algorithm to design the CDS with the most stable secondary structure over all possible candidates coding for a given protein sequence. In addition, Terai et al. have implemented their algorithm as a very useful tool called CDSfold. Both of the above algorithms require $\mathcal{O}(L^3)$ in time and $\mathcal{O}(L^2)$ in space, where L is the length of the CDS to be designed. However, they only optimize the secondary structure stability of the designed CDS without considering its codon usage. To address this issue, Zhang et al. [6] borrowed the idea of lattice parsing from computational linguistics to design the CDS by simultaneously optimizing its secondary structure stability and codon adaptation index (CAI), where CAI is a commonly used measure of codon usage in the host [7]. They first used a deterministic finite-state automaton (DFA) to compactly encode all the CDS candidates and then applied the lattice parsing to find the CDS with the optimal balance between secondary structure stability and CAI in the DFA. The time and space complexities of the algorithm developed by Zhang et al. are $\mathcal{O}(L^3)$ and $\mathcal{O}(L^2)$, respectively. In addition, Zhang

Received 30 April 2024; revised 1 August 2025; accepted 4 August 2025. Date of publication 7 August 2025; date of current version 10 December 2025. This work was supported in part by National Science and Technology Council of Taiwan under Grant NSTC113-2221-E-007-109, and Grant NSTC114-2221-E-007-104-MY2. *(Corresponding author: Chin Lung Lu.)*

The authors are with the Department of Computer Science, National Tsing Hua University, Hsinchu 30013, Taiwan (e-mail: yj5ju0bj@gmail.com; daniel072797@gmail.com; cllu@cs.nthu.edu.tw).

This article has supplementary downloadable material available at https://doi.org/10.1109/TCBBIO.2025.3596771, provided by the authors.

Digital Object Identifier 10.1109/TCBBIO.2025.3596771

et al. [6] further utilized a popular pruning technique, called beam search [12], to significantly reduce the search space of their algorithm without sacrificing much of its accuracy. Zhang et al. have also implemented their algorithm as a tool, called LinearDesign, which can obtain a high-quality approximate CDS in $\mathcal{O}(L)$ time. Unfortunately, however, the beam search function in the current standalone version of LinearDesign is not yet open to users.

Moreover, in the study by Zhang et al. [6], the authors described that due to the usage of extended nucleotides, it is difficult and even impossible for CDSfold to incorporate CAI into the representation of its extended nucleotides such that it can jointly optimize secondary structure stability and CAI when designing a CDS, just as LinearDesign does. The so-called extended nucleotides actually are different notations to represent the same nucleotide at the second position of some amino acids, such as arginine and leucine. They are used by CDSfold to deal with the nucleotide dependencies between different codon positions when executing its dynamic programming algorithm. However, in this study, we have successfully modified the dynamic programming algorithm of CDSfold such that it can do the same job as LinearDesign by jointly optimizing the secondary structure stability and CAI of the CDS to be designed. The key point for this success lies in that we make a simple modification to the extended nucleotides used in CDSfold so that it still can handle the nucleotide dependencies and also integrate CAI into the representation of new extended nucleotides without producing any irreconcilable difference as mentioned by Zhang et al. in their study [6]. The time and space complexities of our improved dynamic programming algorithm are $\mathcal{O}(L^3)$ and $\mathcal{O}(L^2)$, respectively. In addition, by further incorporating the beam search heuristic, our dynamic programming algorithm can obtain an approximate CDS design with high quality in $\mathcal{O}(L)$ time. We have also implemented this dynamic programming algorithm under loop-based energy model into a program called LinearCDSfold.

Note that at the time of writing of this article, Gu et al. [13] have proposed another CDS design tool, called DERNA, which is also based on a dynamic programming algorithm. However, the dynamic programming algorithm of DERNA, which runs in $\mathcal{O}(L^3)$ time and $\mathcal{O}(L^2)$ space, is still distinct from the one we used in LinearCDSfold. To avoid the difficulty caused by CDSfold, DERNA considers three nucleotides within a codon simultaneously, rather than focusing on individual nucleotides as our LinearCDSfold does, during the stepwise process of its dynamic programming algorithm. The benefit to DERNA is that it does not have to handle the nucleotide dependencies of different positions in the codons of arginine and leucine. However, this approach adopted by DERNA comes at the expense of a significantly higher computational cost when compared to our LinearCDSfold, even though both tools share the same time and space complexities. For example, when tested on a protein sequence with an average length of 1,008 amino acids using a 2.4 GHz CPU, our LinearCDSfold takes 30.9 minutes, while DERNA requires 212.7 minutes.

Our experimental results on a real dataset of protein sequences have finally shown that our LinearCDSfold has comparable accuracy to LinearDesign and DERNA in terms of both MFE and CAI, when all these three tools were run with exact search. In terms of running time, however, our LinearCDSfold is slower than LinearDesign, but significantly faster than DERNA, even though both LinearDesign and DERNA share the same time and space complexities as our LinearCDSfold. The results of our experiments have also shown that there is a trade-off between secondary structure stability and codon usage when optimizing their joint optimization objective, meaning that increasing secondary structure stability leads to decreasing codon usage and vice versa. In addition, our LinearCDSfold using beam search can design an approximate CDS with very high quality in very short time. For example, our LinearCDSfold takes only in 1.67 minutes to complete its beam search for a test protein sequence with an average length of 1,008 amino acids.

II. Methods

A. Problem Formulation

In this study, two objectives need to be optimized for designing a CDS for a given amino acid sequence: structural stability and codon usage. The former is to search for the CDS with the lowest minimum-free-energy (MFE) secondary structure among all possible CDSs encoding the given amino acid sequence, while the latter is for the CDS with the maximum codon adaptation index (CAI). Given a CDS $R = r_1 r_2 \dots r_L$, its CAI is defined by Sharp and Li [7] as the geometric mean of the relative adaptiveness values of all codons in R. More specifically, let $C = c_1 c_2 \dots c_l$ denote the codon sequence of R and $c_i = r_{3i-2} r_{3i-1} r_{3i}$ be the ith codon in C, where $l = \frac{L}{3}$ and $1 \leq i \leq l$. In addition, let $w(c_i)$ be the *relative adaptiveness* of the codon c_i that is the frequency of c_i divided by the frequency of its most frequent synonymous codon. The CAI of R is then defined as follows.

$$\mathrm{CAI}(R) = \left(\prod_{1 \leq i \leq l} w(c_i) \right)^{\frac{1}{l}} \tag{1}$$

By the above definition, $\mathrm{CAI}(R)$ is a positive value between 0 and 1. However, the MFE of the most stable secondary structure of R, denoted by $\mathrm{MFE}(R)$, is a negative value proportional to the sequence length of R. Therefore, to reconcile such a difference, Zhang et al. [6] defined a combined optimization objective on R using both $\mathrm{MFE}(R)$ and $\mathrm{CAI}(R)$ as follows.

$$\mathrm{MFECAI}_\lambda(R) = \mathrm{MFE}(R) - \lambda l \log \mathrm{CAI}(R) \tag{2}$$

In (2), λ is a scaling parameter that balances the contributions of $\mathrm{MFE}(R)$ and $\mathrm{CAI}(R)$ to $\mathrm{MFECAI}_\lambda(R)$. In particular, only $\mathrm{MFE}(R)$ is considered if $\lambda = 0$; otherwise, the larger the value of λ, the greater the contribution of $\mathrm{CAI}(R)$. By substituting (1) into (2), the value of $\mathrm{MFECAI}_\lambda(R)$ can be rewritten below as $\mathrm{MFE}(R)$ plus the λ-scaled sum of the negative logarithm of the relative adaptiveness of each codon.

$$\mathrm{MFECAI}_\lambda(R) = \mathrm{MFE}(R) + \lambda \sum_{1 \leq i \leq l} - \log w(c_i) \tag{3}$$

With the joint optimization objective in (3), the CDS design problem we study in this paper can be formulated as follows. Given the target of an amino acid sequence $A = a_1a_2 \dots a_l$ and a non-negative constant λ, the *CDS design problem* is to design a CDS that has the lowest score of MFECAI_λ among all possible CDS candidates encoding A. Recall that the relative adaptiveness function w is defined on codons, not on nucleotides. For designing a dynamic programming algorithm to solve the CDS design problem based on nucleotides rather than codons, we introduce a new function θ to replace the original relative adaptiveness $w(c_i)$ of codon c_i, where $1 \le i \le l$, which is defined on the three nucleotides r_{3i-2}, r_{3i-1} and r_{3i} of c_i such that $\theta(r_{3i-2}) = \theta(r_{3i-1}) = 0$ and $\theta(r_{3i}) = -\log w(c_i)$. With the help of this function θ, the value of $\mathrm{MFECAI}_\lambda(R)$ for the CDS $R = r_1r_2 \dots r_L$ can be further expressed as follows.

$$\mathrm{MFECAI}_\lambda(R) = \mathrm{MFE}(R) + \lambda \sum_{1\le i\le L} \theta(r_i) \tag{4}$$

In the following, we call $\theta(r_i)$ as *CAI score* of nucleotide r_i for simplicity.

B. New Extended Nucleotides

For convenience, we follow the notations used in the study of CDSfold [11] to introduce our algorithms. For each $1 \le i \le L$, let N_i denote the set of possible nucleotides at position i of the CDS $R = r_1r_2 \dots r_L$ to be designed (i.e., $r_i \in N_i$). Basically, according to the genetic code for amino acids, at most 2 nucleotides are allowed to appear in the first and the second codon positions, while at most 4 nucleotides are allowed in the third codon position. In addition, there are the so-called nucleotide dependencies between different codon positions for some amino acids. For instance, amino acid serine is encoded by six codons which are AGC, AGU, UCA, UCG, UCC and UCU. On one hand, if A appears at the first codon position of serine, then G has to appear at its second codon position. On the other hand, if the first codon position of serine is U, then its second codon position must be C. In other words, serine has the nucleotide dependency between its first and second codon positions. Similarly, it can also be found that leucine and arginine both have the nucleotide dependencies between their first and third codon positions. To deal with the dependency between any two consecutive nucleotides, CDSfold introduced two notations $N_i|n$ and $N_i \wedge n$ for correctly designing the CDS, where $N_i|n$ denotes the set of possible nucleotides allowed to appear at position i when the nucleotide at position $i-1$ is n, and $N_i \wedge n$ denotes the set of allowable nucleotides at position i when the nucleotide at position $i+1$ is n. In addition, CDSfold introduced the so-called extended nucleotides to cope with the nucleotide dependencies between the first and the third codon positions for both arginine and leucine. For instance, arginine is encoded by six codons that are AGA, AGG, CGA, CGG, CGC and CGU. Therefore, two kinds of Gs, denoted by $\mathtt{G}^{\mathtt{AG}}$ and $\mathtt{G}^{\mathtt{CU}}$, are created by CDSfold for the second codon position of arginine based on the nucleotide at the third codon position (see Fig. 1, left panel). It means that the nucleotide at the second codon position of arginine is $\mathtt{G}^{\mathtt{AG}}$ (respectively, $\mathtt{G}^{\mathtt{CU}}$) if its nucleotide at the third codon position is

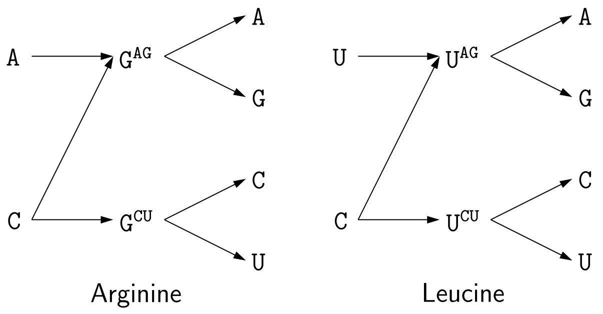

A
C
$\mathtt{G}^{\mathtt{AG}}$
$\mathtt{G}^{\mathtt{CU}}$
A
G
C
U
Arginine
U
C
$\mathtt{U}^{\mathtt{AG}}$
$\mathtt{U}^{\mathtt{CU}}$
A
G
C
U
Leucine

Fig. 1. Representations of extended nucleotides for the second codon position of arginine (left) and leucine (right), where the dependencies of two adjacent nucleotides are indicated by arrows.

A or G (respectively, C or U). Similarly, two kinds of Us, denoted by $\mathtt{U}^{\mathtt{AG}}$ and $\mathtt{U}^{\mathtt{CU}}$, are also used to denote the nucleotide at the second codon position of leucine (see Fig. 1, right panel). Using these extended nucleotides, the dependencies of non-adjacent nucleotides for arginine and leucine can be converted into the adjacent nucleotide dependencies. For instance, if the nucleotide at the first codon position of arginine is A, then the nucleotide at the second codon position must be $\mathtt{G}^{\mathtt{AG}}$, instead of $\mathtt{G}^{\mathtt{CU}}$, which further ensures that the nucleotide at the third codon position must be A or G. On the other hand, if the nucleotide at the first codon position of arginine is C, then the nucleotide at the second codon position can be both $\mathtt{G}^{\mathtt{AG}}$ and $\mathtt{G}^{\mathtt{CU}}$, further implying that the nucleotide at the third codon position can be A, G, C or U.

Note that the joint objective to be optimized in (3) factors the negative logarithm of CAI of a CDS candidate into the negative logarithm of the relative adaptiveness (NLRA for short) of each individual codon. As mentioned in the study of LinearDesign by Zhang et al. [6], it is difficult to integrate such NLRA values of all six codons for arginine and leucine into the representation of their extended nucleotides as shown in Fig. 1. To do this, the NLRA values of the codons must be integrated into the vertices, rather than the edges, in the extended nucleotide representation. Take leucine for example. As given in the study by Zhang et al. [6], the relative adaptiveness values of its six codons UUA, UUG, CUA, CUG, CUC and CUU for human genomes are 0.2, 0.3, 0.2, 1.0, 0.5 and 0.3, respectively and therefore their NLRA values are 0.7, 0.5, 0.7, 0, 0.3 and 0.5, respectively. As shown in Fig. 1 (right panel), the codons $\mathtt{U}\mathtt{U}^{\mathtt{AG}}\mathtt{A}$ and $\mathtt{C}\mathtt{U}^{\mathtt{AG}}\mathtt{A}$ share the same nucleotide A at the third codon position, while $\mathtt{U}\mathtt{U}^{\mathtt{AG}}\mathtt{G}$ and $\mathtt{C}\mathtt{U}^{\mathtt{AG}}\mathtt{G}$ share the same G at the third codon position. Since both $\mathtt{U}\mathtt{U}^{\mathtt{AG}}\mathtt{A}$ and $\mathtt{C}\mathtt{U}^{\mathtt{AG}}\mathtt{A}$ have the same NLRA value of 0.7, their NLRA values can be successfully integrated into the extended nucleotide representation of leucine by assigning the nucleotide A at the third codon position a value equal to 0.7 and the other nucleotides at the first and second codon positions a value of 0. In fact, the values assigned to the nucleotides at the three codon positions are equivalent to the CAI scores of the three nucleotides (i.e., $\theta(r_1), \theta(r_2)$ and $\theta(r_3)$ if the three nucleotides are denoted by r_1, r_2 and r_3). However, the NLRA values of $\mathtt{U}\mathtt{U}^{\mathtt{AG}}\mathtt{G}$ and $\mathtt{C}\mathtt{U}^{\mathtt{AG}}\mathtt{G}$ are different (the former is 0.5 and the latter is 0) and therefore they cannot be simultaneously integrated in the extended nucleotide representation of leucine because

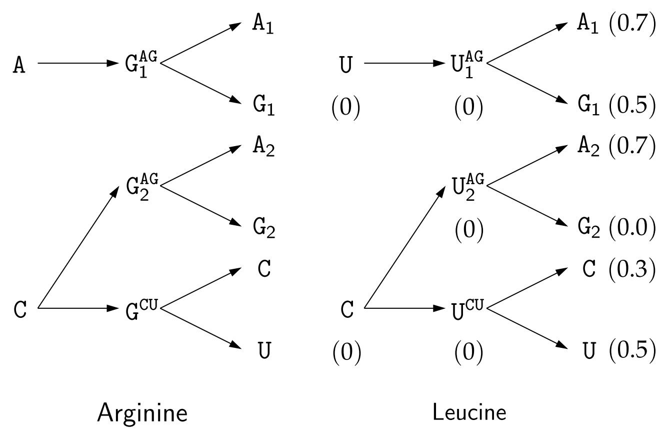


Fig. 2. Representations of new extended nucleotides for the second and third codon positions of arginine (left) and leucine (right), where the arrows depict the dependencies of two adjacent nucleotides. Note that the values in parentheses in the extended nucleotide representation of leucine are the CAI scores of extended nucleotides.

only a single NLRA value (either 0.5 or 0) can be assigned to the nucleotide G at the third codon position. To address the issue mentioned above, we further introduce new extended nucleotides at the second and third codon positions of both arginine and leucine, as shown in Fig. 2. The nucleotide G^{AG} at the second codon position of arginine is further replaced with two kinds of extended nucleotides G_1^{AG} and G_2^{AG}, while the nucleotide U^{AG} at the second codon position of leucine is replaced with U_1^{AG} and U_2^{AG}. Furthermore, two kinds of As, denoted by A_1 and A_2, and two kinds of Gs, denoted by G_1 and G_2, are created to replace the original ones at the third codon position of both arginine and leucine. It means that with respect to arginine and leucine, if G_1^{AG} or U_1^{AG} (respectively, G_2^{AG} or U_2^{AG}) appears at the second codon position, then the nucleotide at the third codon position must be A_1 or G_1 (respectively, A_2 or G_2). Taking leucine as an example, the NLRA values of its six codons can be easily integrated into this new extended nucleotide representation, as shown in Fig. 2 (right panel), by assigning each vertex with the CAI score of its corresponding nucleotide (i.e., $\theta(r_i)$ if the nucleotide is denoted by r_i). In fact, as mention previously, serine has six codons. To properly assign the CAI scores to all the nucleotides in the six codons of serine, two kinds of Cs, denoted by C_C and C_G, and two kinds of Us, denoted by U_C and U_G, are created to replace the original ones at the third codon position (see Fig. S1 in the Supplementary Material for the extended nucleotide representation of serine). It means that if C_C or U_C (respectively, C_G or U_G) appears at the third codon position of serine, then the nucleotide at the second codon position must be C (respectively, G).

C. Algorithm

To simplify the description of our algorithms below, we utilize the base pair-based energy model [14] to measure the free energy of an RNA secondary structure. The free energy of an RNA secondary structure in this simplified model is determined by the sum of the scores of all base pairs in the structure. In this study, the scores of base pairs G-C, A-U and G-U are set as -3, -2 and -1, respectively. In fact, however, our algorithms still can be applied to the loop-based energy model [9] (see the Supplementary Material for details), in which an RNA secondary structure is uniquely decomposed into various loop substructures (e.g., stacked pairs, hairpin loops, bulge loops, internal loops and multi-loops). These loop substructures are assigned energies that are usually established empirically [15]. The free energy of the whole secondary structure is the sum of the energies of all the loop substructures.

Let $\gamma_{i,j}^{n_i,n_j}$ be the minimum score of MFECAI_λ among the subsequences of all CDSs encoding the given amino acid sequence in which each such subsequence starts at position i with nucleotide n_i and ends at position j with nucleotide n_j, where $1 \le i \le j \le L, n_i \in N_i$ and $n_j \in N_j$. Note that if two nucleotides x and y form a base pair (x, y), the function $\delta(x, y)$ is used to represent their base-pairing score; otherwise, $\delta(x, y) = \infty$. Below, we compute each $\gamma_{i,j}^{n_i,n_j}$ by considering what can happen to nucleotide n_j. Case 1: n_j does not form a base pair with other nucleotide between i and j. In this case, if n_{j-1} is in $N_{j-1} \wedge n_j$, then $\gamma_{i,j}^{n_i,n_j}$ equals to $\gamma_{i,j-1}^{n_i,n_{j-1}}$ plus $\lambda\theta(n_j)$, a λ-scaled CAI score of n_j. Case 2: n_j forms a base pair with n_i. Typically, the condition $j - i > t$ must be satisfied, where t is a small positive constant ($t = 3$ by default), because an RNA sequence does not fold back on itself too sharply. For this case, if $n_{i+1} \in N_{i+1}|n_i$ and $n_{j-1} \in N_{j-1} \wedge n_j$, then $\gamma_{i,j}^{n_i,n_j}$ equals to $\gamma_{i+1,j-1}^{n_{i+1},n_{j-1}}$ plus a base-pairing score of $\delta(n_i, n_j)$ plus the sum $\lambda(\theta(n_i) + \theta(n_j))$ of the λ-scaled CAI scores of n_i and n_j. Case 3: n_j forms a base pair with some nucleotide n_k between i and j, where $i < k < j$, $j - k > t$ and $n_k \in N_k$. In this case, the base pair (n_k, n_j) can partition the remaining of the sequence corresponding to $\gamma_{i,j}^{n_i,n_j}$ into two subsequences: (1) the subsequence outside (n_k, n_j) and (2) the subsequence inside (n_k, n_j). If $n_{k-1} \in N_{k-1} \wedge n_k, n_{k+1} \in N_{k+1}|n_k$ and $n_{j-1} \in N_{j-1} \wedge n_j$, then $\gamma_{i,j}^{n_i,n_j}$ equals to $\gamma_{i,k-1}^{n_i,n_{k-1}}$ plus $\gamma_{k+1,j-1}^{n_{k+1},n_{j-1}}$ plus $\delta(n_k, n_j)$ plus $\lambda(\theta(n_k) + \theta(n_j))$. Based on the above discussion, $\gamma_{i,j}^{n_i,n_j}$ can be computed by the following expression.

$$\min \begin{cases} \min\limits_{n_{j-1} \in N_{j-1} \wedge n_j} \gamma_{i,j-1}^{n_i,n_{j-1}} + \lambda\theta(n_j) \\ \min\limits_{\substack{j-i>t \\ n_{i+1} \in N_{i+1}|n_i \\ n_{j-1} \in N_{j-1} \wedge n_j}} \begin{array}{l} \gamma_{i+1,j-1}^{n_{i+1},n_{j-1}} + \delta(n_i, n_j) \\ +\lambda(\theta(n_i) + \theta(n_j)) \end{array} \\ \min\limits_{\substack{i<k<j \\ j-k>t \\ n_k \in N_k \\ n_{k-1} \in N_{k-1} \wedge n_k \\ n_{k+1} \in N_{k+1}|n_k \\ n_{j-1} \in N_{j-1} \wedge n_j}} \begin{array}{l} \gamma_{i,k-1}^{n_i,n_{k-1}} + \gamma_{k+1,j-1}^{n_{k+1},n_{j-1}} \\ +\delta(n_k, n_j) + \lambda(\theta(n_k) + \theta(n_j)) \end{array} \end{cases}$$

With the recursion of $\gamma_{i,j}^{n_i,n_j}$, we design a dynamic programming algorithm as shown in Algorithm 1 to solve the CDS design problem. In Algorithm 1, we calculate $\gamma_{i,j}^{n_i,n_j}$ recursively until the scores of all $\gamma_{1,L}^{n_1,n_L}$ are computed, where $n_1 \in N_1$ and $n_L \in N_L$. Finally, the minimum value of $\gamma_{1,L}^{n_1,n_L}$ (denoted by $\gamma_{1,L}^{\widehat{n_1},\widehat{n_L}}$ in Algorithm 1) among all allowable n_1 and n_L is the minimum MFECAI_λ score over all CDSs encoding the target amino acid.

Algorithm 1: Dynamic Programming Algorithm With Exact Search.

```
1: for $i = 1$ to $L$ do                                                    // Initialization
2:   for each $n_i \in N_i$ do
3:     $\gamma_{i,i}^{n_i,n_i} \leftarrow \lambda\theta(n_i)$
4: for $d = 1$ to $L-1$ do
5:   for $i = 1$ to $L-d$ do
6:     $j \leftarrow i + d$
7:     $\gamma_{i,j}^{n_i,n_j} \leftarrow \infty$ for all $n_i \in N_i$ and $n_j \in N_j$
8:     for each $n_i \in N_i$ and each $n_j \in N_j$ do
9:       for each $n_{j-1} \in N_{j-1} \wedge n_j$ do                       // Unpaired case
10:        $score \leftarrow \gamma_{i,j-1}^{n_i,n_{j-1}} + \lambda\theta(n_j)$
11:        if $\gamma_{i,j}^{n_i,n_j} > score$ then
12:          $\gamma_{i,j}^{n_i,n_j} \leftarrow score$
13:          $backtrace(i,j,n_i,n_j) \leftarrow$ (“unpaired”, $n_{j-1}$)
14:      if $j - i > t$ and $(n_i, n_j)$ can form a base pair then           // Paired case
15:        for each $n_{i+1} \in N_{i+1}|n_i$ and each $n_{j-1} \in N_{j-1} \wedge n_j$ do
16:          $score \leftarrow \gamma_{i+1,j-1}^{n_{i+1},n_{j-1}} + \delta(n_i,n_j) + \lambda(\theta(n_i) + \theta(n_j))$
17:          if $\gamma_{i,j}^{n_i,n_j} > score$ then
18:            $\gamma_{i,j}^{n_i,n_j} \leftarrow score$
19:            $backtrace(i,j,n_i,n_j) \leftarrow$ (“paired”, $(n_{i+1}, n_{j-1})$)
20:      for $k = i+1$ to $j-1$ do                                          // Partition case
21:        if $j - k > t$ then
22:          for each $n_k \in N_k$ and $(n_k, n_j)$ can form a base pair do
23:            for each $n_{k-1} \in N_{k-1} \wedge n_k$, each $n_{k+1} \in N_{k+1}|n_k$ and each $n_{j-1} \in N_{j-1} \wedge n_j$ do
24:              $score \leftarrow \gamma_{i,k-1}^{n_i,n_{k-1}} + \gamma_{k+1,j-1}^{n_{k+1},n_{j-1}} + \delta(n_k,n_j) + \lambda(\theta(n_k) + \theta(n_j))$
25:              if $\gamma_{i,j}^{n_i,n_j} > score$ then
26:                $\gamma_{i,j}^{n_i,n_j} \leftarrow score$
27:                $backtrace(i,j,n_i,n_j) \leftarrow$ (“partition”, $(k, n_{k-1}, n_k, n_{k+1}, n_{j-1})$)
28: Find $\widehat{n_1}$ and $\widehat{n_L}$ such that $\gamma_{1,L}^{\widehat{n_1},\widehat{n_L}} = \min\left\{\gamma_{1,L}^{n_1,n_L} | n_1 \in N_1 \text{ and } n_L \in N_L\right\}$
29: return $\gamma_{1,L}^{\widehat{n_1},\widehat{n_L}}$ and Backtracking$(1, L, \widehat{n_1}, \widehat{n_L})$
```

In addition, an optimal CDS with score $\gamma_{1,L}^{\widehat{n_1},\widehat{n_L}}$ can be obtained by a backtracking procedure as shown in Algorithm 2. The variable $backtrace(i, j, n_i, n_j)$ is to keep track of the information about which of the three cases leads to the value of $\gamma_{i,j}^{n_i,n_j}$ during the dynamic programming process of Algorithm 1. Algorithm 2 then utilizes the backtracking information stored in $backtrace(i, j, n_i, n_j)$ to recover the nucleotide sequence of an optimal CDS and its corresponding secondary structure. Note that the variable $stack$ used in Algorithm 2 is a data structure of stack whose elements are in the form of (i, j, n_i, n_j). It can be verified that the running time of Algorithm 2 is linearly related to the number of push and pop operations and moreover the number of pushes equals to that of pops. In other words, the time complexity of Algorithm 2 is proportional to the number of pushes. Except for prefix case, which is only used in Algorithm 3 introduced later, the other three cases all recover one or two nucleotides of the optimal CDS at the cost of generating one or two push operations. Hence, the number of pushes is linearly proportional to the length L of the CDS to be designed and, as a result, the time complexity of Algorithm 2 is $\mathcal{O}(L)$. Due to the extended nucleotides used in this study, we have $|N_i| \le 6$ for each $1 \le i \le L$ in Algorithm 1. The time complexity of Algorithm 1 clearly is dominated by the nested **for** loops of lines 4–27, which take $\mathcal{O}(L^3)$ time in total. In addition, it is not hard to verify that the space complexity of Algorithm 1 is $\mathcal{O}(L^2)$.

Next, inspired by the study of LinearFold [12], we further accelerate the computation of Algorithm 1 by pruning its search space using the so-called beam search method, a popular heuristic to search for near-optimal solutions of combinatorial optimization problems within a limited time. The basic idea of applying the beam search to our algorithm, as presented in Algorithm 3, is described as follows. Let $\Gamma_{i,j}^{n_i,n_j}$ denote an approximate value of $\gamma_{i,j}^{n_i,n_j}$ and let $\mu = \max_{1 \le i \le L} |N_i|$ (actually $\mu = 6$). Note that Γ is implemented in Algorithm 3 as a hash table that uses (i, j, n_i, n_j) as a key and $\Gamma_{i,j}^{n_i,n_j}$ as the value. For each step j $(2 \le j \le L)$, not all the values of $\Gamma_{i,j}^{n_i,n_j}$ $(1 \le i < j \le L, n_i \in N_i$ and $n_j \in N_j)$ are computed in

Algorithm 2: Backtracking(i, j, n_i, n_j).

```
1: stack ← Push($(i, j, n_i, n_j)$)   // Push $(i, j, n_i, n_j)$ onto stack
2: while stack is not empty do
3:     $(i, j, n_i, n_j)$ ← Pop(stack)   // Pop the top element of stack to $(i, j, n_i, n_j)$
4:     (case, backtrace-information) ← backtrace$(i, j, n_i, n_j)$
5:     if $i = j$ then
6:         structure[$j$] ← "." and cds[$j$] ← $n_j$
7:     else if case = "unpaired" then
8:         structure[$j$] ← "." and cds[$j$] ← $n_j$
9:         $n_{j-1}$ ← backtrace-information
10:        stack ← Push$(i, j-1, n_i, n_{j-1})$
11:    else if case = "paired" then
12:        structure[$i$] ← "(" and structure[$j$] ← ")"
13:        cds[$i$] ← $n_i$ and cds[$j$] ← $n_j$
14:        $(n_{i+1}, n_{j-1})$ ← backtrace-information
15:        stack ← Push$(i+1, j-1, n_{i+1}, n_{j-1})$
16:    else if case = "partition" then
17:        $(k, n_{k-1}, n_k, n_{k+1}, n_{j-1})$ ← backtrace-information
18:        structure[$k$] ← "(" and structure[$j$] ← ")"
19:        cds[$k$] ← $n_k$ and cds[$j$] ← $n_j$
20:        stack ← Push$(i, k-1, n_i, n_{k-1})$ and
            stack ← Push$(k+1, j-1, n_{k+1}, n_{j-1})$
21:    else if case = "prefix" then   // For Algorithm 3
22:        $(i, n_1, n_{i-1}, n_i, n_j)$ ← backtrace-information
23:        stack ← Push$(1, i-1, n_1, n_{i-1})$ and
            stack ← Push$(i, j, n_i, n_j)$
24: return cds and structure
```

Algorithm 3, but at most only the $b + \mu$ such values with the possible lowest scores need to be computed, where b is the so-called beam size predefined by the user. Moreover, these scores of $\Gamma_{i,j}^{n_i,n_j}$ are obtained by extending the approximate scores already computed in the previous steps. For this purpose, three kinds of extensions are considered below to obtain the score of $\Gamma_{i,j}^{n_i,n_j}$. Case 1: $n_j \in N_j, n_{j-1} \in N_{j-1} \wedge n_j$ and $\Gamma_{i,j-1}^{n_i,n_{j-1}} \neq$ NULL (i.e., $\Gamma_{i,j-1}^{n_i,n_{j-1}}$ has been updated in step $j-1$). In this case, $\Gamma_{i,j}^{n_i,n_j}$ is updated by $\Gamma_{i,j-1}^{n_i,n_{j-1}} + \lambda\theta(n_j)$ if the former is NULL or its value is greater than the latter. Case 2: $j - i > t$, both $n_i \in N_i$ and $n_j \in N_j$ form a base pair, and $\Gamma_{i+1,j-1}^{n_{i+1},n_{j-1}} \neq$ NULL. Then $\Gamma_{i,j}^{n_i,n_j}$ is updated by $\Gamma_{i+1,j-1}^{n_{i+1},n_{j-1}} + \delta(n_i, n_j) + \lambda(\theta(n_i) + \theta(n_j))$ if the former is NULL or its value is greater than the latter. Case 3: $j - k > t$, both $n_k \in N_k$ and $n_j \in N_j$ form a base pair, $\Gamma_{i,k-1}^{n_i,n_{k-1}} \neq$ NULL and $\Gamma_{k+1,j-1}^{n_{k+1},n_{j-1}} \neq$ NULL. In this case, $\Gamma_{i,j}^{n_i,n_j}$ is updated by $\Gamma_{i,k-1}^{n_i,n_{k-1}} + \Gamma_{k+1,j-1}^{n_{k+1},n_{j-1}} + \delta(n_k, n_j) + \lambda(\theta(n_k) + \theta(n_j))$ if the former is NULL or its value is greater than the latter. Since the above three cases are similar to those in Algorithm 1, we call them "unpaired", "paired" and "partition" extensions, respectively. After these three extensions are done, all the scores of the updated $\Gamma_{i,j}^{n_i,n_j}$ are ranked from small to large by the sum of $\Gamma_{1,i-1}^{n_1,n_{i-1}}$ and $\Gamma_{i,j}^{n_i,n_j}$ (called *prefix score*), provided that $n_{i-1} \in N_{i-1} \wedge n_i$ and $\Gamma_{1,i-1}^{n_1,n_{i-1}} \neq$ NULL. Next, at most the b updated $\Gamma_{i,j}^{n_i,n_j}$ with the lowest prefix scores are kept and the others are pruned away because their corresponding subsequences are less likely to be part of the optimal CDS. The pseudocode of the above beam pruning process is presented in Algorithm 4. After the pruning process, each unpruned $\Gamma_{i,j}^{n_i,n_j}$ is further combined with $\Gamma_{1,i-1}^{n_1,n_{i-1}}$ to obtain $\Gamma_{1,j}^{n_1,n_j}$ with lower score, provided that $\Gamma_{1,i-1}^{n_1,n_{i-1}} \neq$ NULL. More specifically, $\Gamma_{1,j}^{n_1,n_j}$ is updated by $\Gamma_{1,i-1}^{n_1,n_{i-1}} + \Gamma_{i,j}^{n_i,n_j}$ if the former is NULL or its value is greater than the latter. We call the above process "prefix" extension because the obtained $\Gamma_{1,j}^{n_1,n_j}$ is an approximation of $\gamma_{1,j}^{n_1,n_j}$ corresponding to a prefix subsequence of the optimal CDS. In addition, the score of $\Gamma_{1,j}^{n_1,n_j}$ can be used to calculate the prefix score of $\Gamma_{j+1,j'}^{n_{j+1},n_{j'}}$ extended in a later step, where $j < j' \leq L$. During the prefix extension, at most μ scores of $\Gamma_{1,j}^{n_1,n_j}$ are generated (since $|N_1| = 1$ and $|N_j| \leq 6$), leading to at most $b + \mu$ non-null $\Gamma_{i,j}^{n_i,n_j}$ are produced in the jth step. Finally, Algorithm 3 outputs the minimum value of non-null $\Gamma_{1,L}^{n_1,n_L}$ among all allowable n_1 and n_L and its corresponding CDS obtained by using the backtracking procedure in Algorithm 2.

The time complexity of Algorithm 3 is analyzed as follows. The initialization at lines 2–4 can be done in $\mathcal{O}(L)$ time. Recall that, due to the beam search, each step j in Algorithm 3 generates up to $b + \mu$ non-null $\Gamma_{i,j}^{n_i,n_j}$, where $2 \leq j \leq L$. There are $\mathcal{O}((b+\mu)\mu) = \mathcal{O}(b\mu)$ iterations of the two nested **for** loops of lines 6–12, each of which takes constant time. Hence, the unpaired extension can be done in $\mathcal{O}(b\mu)$ time. Because the two nested **for** loops of lines 13–20 have $\mathcal{O}((b+\mu)\mu^2) = \mathcal{O}(b\mu^2)$ iterations and each iteration takes constant time, the paired extension costs $\mathcal{O}(b\mu^2)$ time. The partition extension in lines 21–29 requires time $\mathcal{O}(b^2\mu^2)$, since its three nested **for** loops iterate $\mathcal{O}((b+\mu)^2\mu^2) = \mathcal{O}(b^2\mu^2)$ times and the running time of each iteration is constant. Note that the above three extensions generate $\mathcal{O}(b^2\mu^2)$ non-null $\Gamma_{i,j}^{n_i,n_j}$. As for the procedure BeamPrune(j, b) called by Algorithm 3, a total of $\mathcal{O}(b^2\mu^2)$ candidates need to be considered, since the outer **for** loop beginning in line 2 of Algorithm 4 contains $\mathcal{O}(b^2\mu^2)$ iterations, each of which generates a candidate in $\mathcal{O}(\mu)$ time (since the inner **for** loop of lines 7–9 has $\mathcal{O}(\mu)$ iterations). The selection of the bth lowest score from $\mathcal{O}(b^2\mu^2)$ candidates in line 11 actually can be done in linear time [16] and the beam pruning process of lines 12–15 takes $\mathcal{O}(b^2\mu^2)$ time. Hence, the total running time of BeamPrune(j, b) is $\mathcal{O}(b^2\mu^3)$. For the prefix extension in Algorithm 3, it runs in $\mathcal{O}(b\mu^2)$ time, since its two nested `for` loops in lines 31–37 contains $\mathcal{O}(b\mu^2)$ iterations and each of such iterations takes constant time. As a result, the execution of lines 5–37 in Algorithm 3 takes $\mathcal{O}(b^2\mu^3L)$ time. Clearly, line 38 in Algorithm 3 takes $\mathcal{O}(\mu)$ time. As analyzed before, the backtracking procedure in line 39 still runs in $\mathcal{O}(L)$ time, even if it needs the additional prefix case to recover the nucleotides of the CDS to be designed. The reason is that the number of the prefix cases needed in the backtracking procedure is linearly proportional to the length L of the CDS and each prefix case generates two push operations. Based on the discussion above, therefore, the time complexity of Algorithm 3 is $\mathcal{O}(b^2\mu^3L) = \mathcal{O}(L)$ (since both b and μ are constants). On the other hand, it can be verified that the space complexity of Algorithm 3 still is $\mathcal{O}(L)$.

The dynamic programming algorithm with exact search and its beam pruning version we described above utilize the base pair-based energy model to measure the folding energy of

Algorithm 3: Dynamic Programming Algorithm With Beam Search.

```
1: $\Gamma \leftarrow$ hash() and $backtrace \leftarrow$ hash()        // Initialize $\Gamma$ and $backtrace$ as hash tables
2: for $i = 1$ to $L$ do
3:   for each $n_i \in N_i$ do        // Initialization
4:     $\Gamma_{i,i}^{n_i,n_i} \leftarrow \lambda\theta(n_i)$
5: for $j = 2$ to $L$ do
6:   for each $(i, j-1, n_i, n_{j-1})$ such that $\Gamma_{i,j-1}^{n_i,n_{j-1}} \neq$ NULL do        // Unpaired extension
7:     for each $n_j \in N_j$ do
8:       if $n_{j-1} \in N_{j-1} \wedge n_j$ then
9:         $score \leftarrow \Gamma_{i,j-1}^{n_i,n_{j-1}} + \lambda\theta(n_j)$
10:        if $\Gamma_{i,j}^{n_i,n_j}$ = NULL or $\Gamma_{i,j}^{n_i,n_j} > score$ then        // Update $\Gamma_{i,j}^{n_i,n_j}$
11:          $\Gamma_{i,j}^{n_i,n_j} \leftarrow score$
12:          $backtrace(i, j, n_i, n_j) \leftarrow$ ("unpaired", $n_{j-1}$)
13:   for each $(i+1, j-1, n_{i+1}, n_{j-1})$ such that $\Gamma_{i+1,j-1}^{n_{i+1},n_{j-1}} \neq$ NULL do        // Paired extension
14:     if $j - i > t$ then
15:       for each $n_i \in N_i$, each $n_j \in N_j$ and $(n_i, n_j)$ can form a base pair do
16:         if $n_{i+1} \in N_{i+1}|n_i$ and $n_{j-1} \in N_{j-1} \wedge n_j$ then
17:           $score \leftarrow \Gamma_{i+1,j-1}^{n_{i+1},n_{j-1}} + \delta(n_i, n_j) + \lambda(\theta(n_i) + \theta(n_j))$
18:           if $\Gamma_{i,j}^{n_i,n_j}$ = NULL or $\Gamma_{i,j}^{n_i,n_j} > score$ then
19:             $\Gamma_{i,j}^{n_i,n_j} \leftarrow score$
20:             $backtrace(i, j, n_i, n_j) \leftarrow$ ("paired", $(n_{i+1}, n_{j-1})$)
21:   for each $(k+1, j-1, n_{k+1}, n_{j-1})$ such that $\Gamma_{k+1,j-1}^{n_{k+1},n_{j-1}} \neq$ NULL do        // Partition extension
22:     if $j - k > t$ then
23:       for each $n_k \in N_k$, each $n_j \in N_j$ and $(n_k, n_j)$ can form a base pair do
24:         for each $(i, k-1, n_i, n_{k-1})$ such that $\Gamma_{i,k-1}^{n_i,n_{k-1}} \neq$ NULL do
25:           if $n_{k-1} \in N_{k-1} \wedge n_k$, $n_{k+1} \in N_{k+1}|n_k$ and $n_{j-1} \in N_{j-1} \wedge n_j$ then
26:             $score \leftarrow \Gamma_{i,k-1}^{n_i,n_{k-1}} + \Gamma_{k+1,j-1}^{n_{k+1},n_{j-1}} + \delta(n_k, n_j) + \lambda(\theta(n_k) + \theta(n_j))$
27:             if $\Gamma_{i,j}^{n_i,n_j}$ = NULL or $\Gamma_{i,j}^{n_i,n_j} > score$ then
28:               $\Gamma_{i,j}^{n_i,n_j} \leftarrow score$
29:               $backtrace(i, j, n_i, n_j) \leftarrow$ ("partition", $(k, n_{k-1}, n_k, n_{k+1}, n_{j-1})$)
30:   BeamPrune$(j, b)$        // $b$ is the beam size
31:   for each $(i, j, n_i, n_j)$ such that $i \neq 1$ and $\Gamma_{i,j}^{n_i,n_j} \neq$ NULL do        // Prefix extension
32:     for each $n_1 \in N_1$ and each $n_{i-1} \in N_{i-1}$ do
33:       if $n_{i-1} \in N_{i-1} \wedge n_i$ and $\Gamma_{1,i-1}^{n_1,n_{i-1}} \neq$ NULL then
34:         $score \leftarrow \Gamma_{1,i-1}^{n_1,n_{i-1}} + \Gamma_{i,j}^{n_i,n_j}$
35:         if $\Gamma_{1,j}^{n_1,n_j}$ = NULL or $\Gamma_{1,j}^{n_1,n_j} > score$ then
36:           $\Gamma_{1,j}^{n_1,n_j} \leftarrow score$
37:           $backtrace(1, j, n_1, n_j) \leftarrow$ ("prefix", $(i, n_1, n_{i-1}, n_i, n_j)$)
38: Find $\widehat{n_1}$ and $\widehat{n_L}$ such that $\Gamma_{1,L}^{\widehat{n_1},\widehat{n_L}} = \min\left\{\Gamma_{1,L}^{n_1,n_L} | n_1 \in N_1, n_L \in N_L \text{ and } \Gamma_{1,L}^{n_1,n_L} \neq \text{NULL}\right\}$
39: return $\Gamma_{1,L}^{\widehat{n_1},\widehat{n_L}}$ and Backtracking$(1, L, \widehat{n_1}, \widehat{n_L})$
```

an RNA secondary structure, but they remain effective even when employing the loop-based energy model for measuring folding energy (see the Supplementary Material for details). Finally, we have used C++ to implement these two algorithms on the loop-based energy model into a program, called LinearCDSfold, whose source code is freely available at https://github.com/ablab-nthu/LinearCDSfold.

III. Results and Discussion

In the following, we evaluated our CDS design tool LinearCDSfold with different parameter settings on a number of protein sequences collected in the UniProt [17] database. In addition, we compared its performance with those obtained by CDSfold [11], LinearDesign [6] and DERNA [13] in terms of MFE,

Algorithm 4: BeamPrune(j, b).

```
1: $candidates \leftarrow$ hash()  // Initialize $candidates$ as a hash table
2: for each $(i, j, n_i, n_j)$ such that $\Gamma_{i,j}^{n_i,n_j} \neq$ NULL do
3:   if $i = 1$ then
4:     $candidates(i, j, n_i, n_j) \leftarrow \Gamma_{i,j}^{n_i,n_j}$
5:   else
6:     $bestscore \leftarrow 0$
7:     for each $(1, i-1, n_1, n_{i-1})$ such that $\Gamma_{1,i-1}^{n_1,n_{i-1}} \neq$ NULL do
8:       if $n_{i-1} \in N_{i-1} \wedge n_i$ and $bestscore > \Gamma_{1,i-1}^{n_1,n_{i-1}} + \Gamma_{i,j}^{n_i,n_j}$ then
9:         $bestscore \leftarrow \Gamma_{1,i-1}^{n_1,n_{i-1}} + \Gamma_{i,j}^{n_i,n_j}$
10:    $candidates(i, j, n_i, n_j) \leftarrow bestscore$
11: $bscore \leftarrow$ select the $b$th lowest score in $candidates$
12: for each $(i, j, n_i, n_j)$ such that $\Gamma_{i,j}^{n_i,n_j} \neq$ NULL do
13:   if $candidates(i, j, n_i, n_j) > bscore$ then
14:     $\Gamma_{i,j}^{n_i,n_j} \leftarrow$ NULL
15:     $backtrace(i, j, n_i, n_j) \leftarrow$ NULL
```

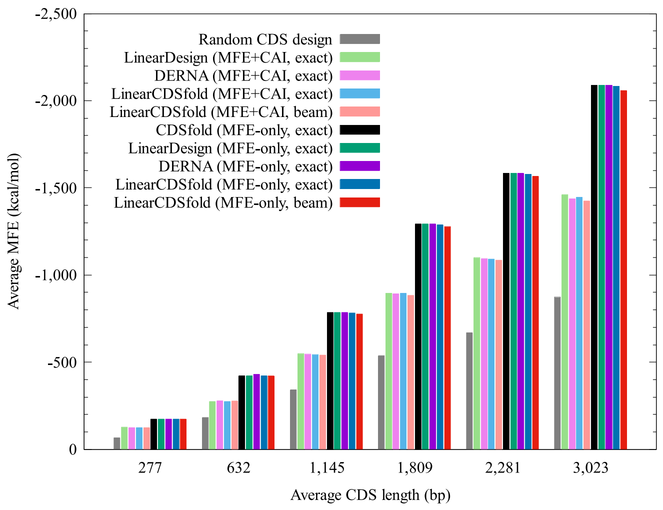


Fig. 3. Comparison of average MFE performance of the evaluated CDS design tools. Basically, the MFE histograms of all the evaluated tools outperform that of the random CDS design. In the MFE-only experiments, the MFE histograms of CDSfold, LinearDesign, DERNA and our LinearCDSfold have highly similar bar heights. Similarly, in the experiments that consider both MFE and CAI, the MFE histograms of LinearDesign, DERNA and our LinearCDSfold closely align in their bar heights. In addition, the MFE histogram of LinearCDSfold with beam search remains very close to that of LinearCDSfold without it, regardless of whether CAI is considered.

CAI and running time. All the experiments were performed on a Linux PC with an 8-core Intel Xeon 2.4 GHz CPU and 32 GB RAM. Note that, at the time of writing this paper, the standalone version of LinearDesign has not yet offered the ordinary users the ability to use the function of beam search for an approximate CDS design. Therefore, we only evaluated the exact CDS design of LinearDesign in this study. In addition, the objective function $\mathrm{MFECAI}_\lambda(R)$ defined by DERNA to design an optimal CDS R is a little different from the one we used in this study, which becomes to minimize $\lambda_{\mathrm{DN}}\mathrm{MFE}(R) - (1 - \lambda_{\mathrm{DN}})l \log \mathrm{CAI}(R)$, where $\lambda_{\mathrm{DN}} \in [0, 1]$ is the scaling parameter used by DERNA to balance the contributions of $\mathrm{MFE}(R)$ and $\mathrm{CAI}(R)$, respectively. The amino acid sequences in the test dataset were divided into six intervals of different lengths, namely $[1, 100]$, $[101, 300]$, $[301, 500]$, $[501, 700]$, $[701, 900]$ and $[901, 1100]$ amino acids, and for each interval length, we randomly selected 10 protein sequences from the UniProt database (see Table S1 in the Supplementary Material for their details). For the purpose of comparison, a random CDS was generated by a method that randomly selects an available codon for each amino acid in each test protein sequence. For the fair comparison, the MFE values of all CDSs to be compared were recalculated using RNAfold [18]. Moreover, all the evaluated CDS design tools used the codon frequencies of *Saccharomyces cerevisiae* reported in the Codon Usage Database [19] to calculate their CAI values.

Fig. 3 depicts the MFE histograms generated by all the CDS design tools evaluated in this study when using different parameter settings, as well as the MFE histogram of randomly designed CDS sequences, where each bar in the histogram represents an average from 10 different samples. For a more detailed list of their MFE values, see Table S2 in the Supplementary Material. Note that we failed to obtain any result when applying DERNA to a test protein sequence (UniProt ID P0AC51) with 171 amino acids, regardless of whether CAI was considered or not, and hence the second bar in the MFE histogram of DERNA, as well as in its CAI and running time histograms discussed later, was obtained by averaging across the remaining nine samples with a length in the range [301,500]. As depicted in Fig. 3, the MFE histograms of all the evaluated tools are better than the one of the random CDS design. On the other hand, when CAI is not considered in the joint optimization objective MFECAI_λ (i.e., $\lambda = 0$ and $\lambda_{\mathrm{DN}} = 1$ or equivalently MFE-only), the MFE histograms of CDSfold, LinearDesign, DERNA and our LinearCDSfold show highly similar bar heights and they are the best among all the ten MFE histograms to be compared. In the MFE-only version, moreover, the MFE histogram for LinearCDSfold running with beam search of size $b = 500$ is very close to the MFE histogram for LinearCDSfold when using exact search, suggesting that the beam search quality of our LinearCDSfold is very high when assessed in terms of MFE. This property persists even when CAI is also taken into account in the optimization objective. For example, when using $\lambda = 3$, the histograms of LinearCDSfold with exact search and with beam search ($b = 500$) still show similar bar heights. In this case, where CAI is also taken into account, the MFE histograms of LinearDesign, DERNA and our LinearCDSfold align closely in their bar heights, when running the exact search with $\lambda = 3$ and $\lambda_{\mathrm{DN}} = 0.00325$. However, their MFE performances decrease when compared to those obtained without considering CAI. This is because these three tools need to sacrifice some MFE values to optimize their MFECAI_λ values.

Fig. 4 illustrates the CAI histograms generated by all the evaluated CDS design tools, considering different parameter settings, and the randomly designed CDS sequences, where in principle each bar in a CAI histogram represents an average from

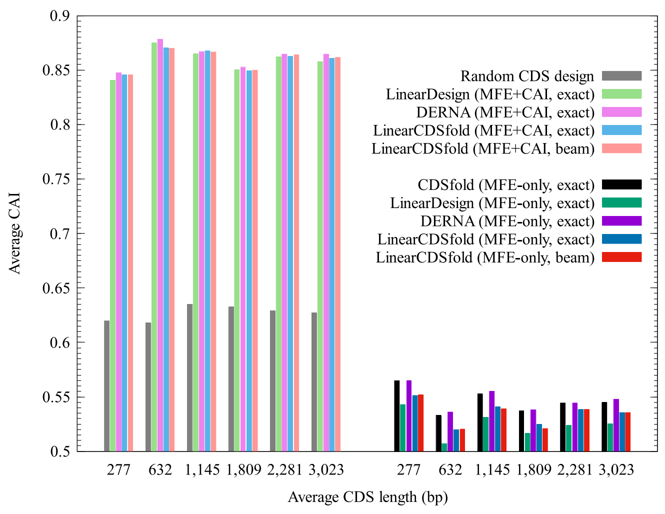


Fig. 4. Comparison of average CAI performance of the evaluated CDS design tools. In the MFE-only experiments, the CAI histograms of CDSfold, LinearDesign, DERNA and our LinearCDSfold are all inferior to that of the random CDS design. However, the CAI histograms of our LinearCDSfold, whether obtained through exact search or beam search, are close to each other in their bar heights and, moreover, are superior to those of LinearDesign, despite remaining inferior to those of CDSfold and DERNA. In the experiments considering both MFE and CAI, the CAI histograms of LinearDesign, DERNA and our LinearCDSfold perform competitively with one another and are significantly better than that of the random CDS design. Moreover, the CAI histograms of our LinearCDSfold obtained with exact search and beam search respectively have highly similar bar heights.

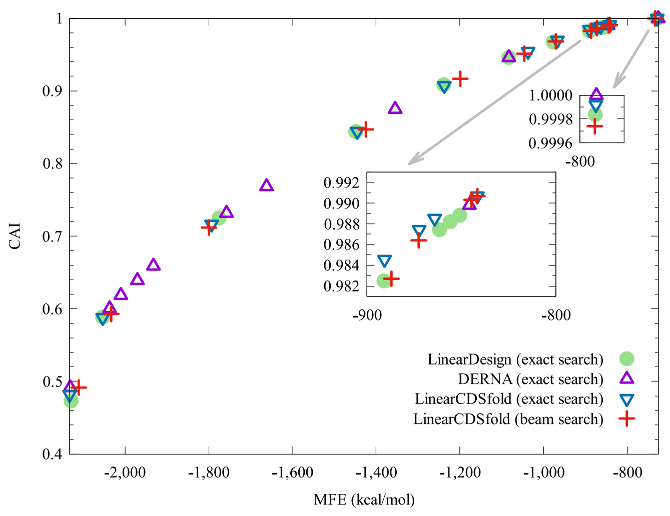


Fig. 5. Comparison of CAI vs. MFE curves (shown as discrete points) for LinearDesign, DERNA and LinearCDSfold when applied to a test protein sequence (UniProt ID Q2QGD7). The CAI vs. MFE curves of LinearDesign, DERNA and our LinearCDSfold closely resemble one another. Furthermore, the CAI vs. MFE curve generated by our LinearCDSfold using beam search closely aligns with the one produced by exact search.

10 different samples. For a more detailed list of their CAI values, see Table S3 in the Supplementary Material. As illustrated in Fig. 4, the CAI performances of the evaluated CDS design tools, including CDSfold, LinearDesign (with $\lambda = 0$ and exact search), DERNA (with $\lambda_{DN} = 1$ and exact search), LinearCDSfold (with $\lambda = 0$ and exact search) and LinearCDSfold (with $\lambda = 0$ and beam search of size $b = 500$), are all inferior to that of the random CDS design. This is because CAI is not taken into account in the objective function of these tools. In this case (where CAI is not considered), however, the CAI histograms of our LinearCDSfold, whether obtained through exact search or beam search (with size $b = 500$), are close to each other in their bar heights and moreover they are superior to that of LinearDesign, although they remain inferior to those of CDSfold and DERNA. On the other hand, when CAI is also considered in the objective (e.g., $\lambda = 3$ for LinearDesign and LinearCDSfold and $\lambda_{DN} = 0.00325$ for DERNA), the CAI histograms of LinearDesign, DERNA and our LinearCDSfold, all running with exact search, compete well with one another and are significantly superior to the one of the random CDS design. In addition, the CAI histograms of our LinearCDSfold obtained with exact search and beam search ($b = 500$) respectively exhibit highly similar bar heights, suggesting that the beam search quality of our LinearCDSfold is also very high when assessed in terms of CAI.

According to the discussion above, our LinearCDSfold appears to offer comparable accuracy to LinearDesign and DERNA in terms of both MFE and CAI. To further validate this characteristic, we plot the CAI versus MFE curves, shown as discrete points, for LinearDesign, DERNA and our LinearCDSfold when applied to a protein sequence (UniProt ID Q2QGD7) in the test dataset by using different values for λ and λ_{DN}, where specifically $\lambda \in \{0, 1, 2, \ldots, 10, 30\}$ and $\lambda_{DN} \in \{0, 0.001, 0.002, \ldots, 0.009, 1\}$. As a result, the CAI vs. MFE curves of LinearDesign, DERNA and LinearCDSfold closely resemble one another as shown in Fig. 5, confirming that our LinearCDSfold provides similar accuracy to LinearDesign and DERNA in terms of both MFE and CAI, even though their algorithms and implementations are totally different. In addition, the curves in Fig. 5 depict a trade-off between MFE (i.e., structural stability) and CAI (i.e., codon usage) when optimizing $\mathrm{MFECAI}_{\lambda}$. In other words, increasing MFE leads to decreasing CAI and vice versa. On the other hand, the CAI vs. MFE curve obtained by our LinearCDSfold using beam search ($b = 500$) is very close to that produced by exact search, suggesting that the beam search quality of LinearCDSfold is exceptionally high when evaluated in terms of both MFE and CAI.

Fig. 6 shows the running time histograms of all the evaluated CDS design tools with different parameter settings, where in principle each bar in a histogram represents an average running time from 10 different samples. For a more detailed list of their running time, see Table S4 in the Supplementary Material. As shown in Fig. 6, the running time histogram of our LinearCDSfold, when running with MFE only and exact search, closely resembles that of CDSfold. This similarity arises because both of them were implemented based on similar dynamic programming techniques. As compared to DERNA and LinearDesign, the exact-search running time of our LinearCDSfold is higher than that of LinearDesign, but significantly lower than that of DERNA, regardless of whether CAI is considered or not. Taking a test protein sequence with an average length of 1,008 amino acids as an example, and considering both MFE and CAI during optimization, LinearDesign and our LinearCDSfold take

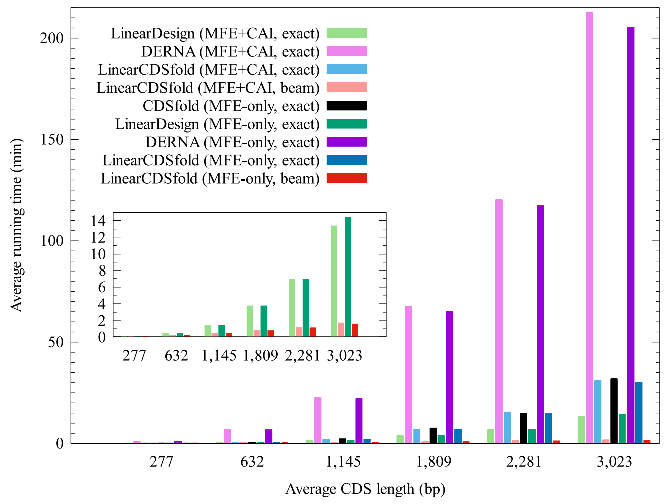


Fig. 6. Comparison of average running time of the evaluated CDS design tools. The running time histogram of our LinearCDSfold, when operating with MFE only and exact search, closely resembles that of CDSfold. As compared to DERNA and LinearDesign, the exact-search running time of our LinearCDSfold is higher than that of LinearDesign, but significantly lower than that of DERNA, regardless of whether CAI is considered. Furthermore, the running time of our LinearCDSfold using beam search is substantially lower than that of other tools running with exact search.

13.4 and 30.9 minutes, respectively, while DERNA requires 212.7 minutes. It indicates that our LinearCDSfold exhibits much greater efficiency than DERNA, even though both of their dynamic programming algorithms scale cubically with the length of the CDS to be designed. On the other hand, however, the running time of our LinearCDSfold when using beam search is more significantly lower than that of other tools running with exact search. For example, our LinearCDSfold completes its beam search with high quality only in 1.67 minutes for a test protein sequence with an average length of 1,008 amino acids.

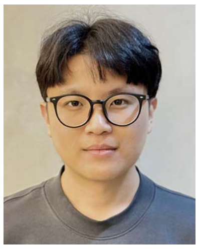
IV. Conclusion

In this paper, we studied how to modify the dynamic programming algorithm of CDSfold such that it can solve the CDS design problem, which is to design a CDS with the lowest score of MFECAI_λ among all possible CDS candidates encoding a given amino acid sequence. Even though this task has been considered a difficult challenge, we have successfully modified the dynamic programming algorithm of CDSfold such that it can exactly solve the CDS design problem in $\mathcal{O}(L^3)$ time and $\mathcal{O}(L^2)$ space, where L is the length of the CDS to be designed. Furthermore, by incorporating the beam search technique, our modified dynamic programming algorithm can compute an approximate CDS design with high quality in $\mathcal{O}(L)$ time. We have also implemented our dynamic programming algorithm with both exact search and beam search on the loop-energy model into the program LinearCDSfold. Our experimental results on a dataset of protein sequences showed that our LinearCDSfold has comparable accuracy to the state-of-the-art CDS design tools LinearDesign and DERNA in terms of both MFE and CAI, when all these three tools were run with exact search. In addition, the results in our experiments showed that there is a trade-off between MFE (i.e., structural stability) and CAI (i.e., codon usage) when optimizing MFECAI_λ, meaning that increasing MFE leads to decreasing CAI and vice versa. Our experimental results also showed that LinearCDSfold using beam search can design an approximate CDS in a very short time (e.g., within 1.67 minutes for a test protein sequence with an average length of 1,008 amino acids) with very high quality in both MFE and CAI. It is worth noting that LinearDesign has not yet made the function of its beam search available to the ordinary users in its standalone version. Therefore, we believe that our LinearCDSfold can serve as a useful alternative tool to help people design their own CDS sequences when trying to optimize the structural stability and codon usage of the designed CDS sequences simultaneously.

References

[1] F. P. Polack et al., “Safety and efficacy of the BNT162b2 mRNA Covid-19 vaccine,” *New England J. Med.*, vol. 383, pp. 2603–2615, 2020.
[2] L. R. Baden et al., “Efficacy and safety of the mRNA-1273 SARS-CoV-2 vaccine,” *New England J. Med.*, vol. 384, pp. 403–416, 2021.
[3] D. M. Mauger et al., “mRNA structure regulates protein expression through changes in functional half-life,” *Proc. Nat. Acad. Sci. USA*, vol. 116, pp. 24075–24083, 2019.
[4] H. K. Wayment-Steele et al., “Theoretical basis for stabilizing messenger RNA through secondary structure design,” *Nucleic Acids Res.*, vol. 49, pp. 10604–10617, 2021.
[5] K. Leppek et al., “Combinatorial optimization of mRNA structure, stability, and translation for RNA-based therapeutics,” *Nature Commun.*, vol. 13, 2022, Art. no. 1536.
[6] H. Zhang et al., “Algorithm for optimized mRNA design improves stability and immunogenicity,” *Nature*, vol. 621, pp. 396–403, 2023.
[7] P. M. Sharp and W. H. Li, “The codon adaptation index–a measure of directional synonymous codon usage bias, and its potential applications,” *Nucleic Acids Res.*, vol. 15, pp. 1281–1295, 1987.
[8] C. Gustafsson, S. Govindarajan, and J. Minshull, “Codon bias and heterologous protein expression,” *Trends Biotechnol.*, vol. 22, pp. 346–353, 2004.
[9] M. Zuker and P. Stiegler, “Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information,” *Nucleic Acids Res.*, vol. 9, pp. 133–148, 1981.
[10] B. Cohen and S. Skiena, “Natural selection and algorithmic design of mRNA,” *J. Comput. Biol.*, vol. 10, pp. 419–432, 2003.
[11] G. Terai, S. Kamegai, and K. Asai, “CDSfold: An algorithm for designing a protein-coding sequence with the most stable secondary structure,” *Bioinformatics*, vol. 32, pp. 828–834, 2016.
[12] L. Huang et al., “LinearFold: Linear-time approximate RNA folding by 5’-to-3’ dynamic programming and beam search,” *Bioinformatics*, vol. 35, pp. I295–I304, 2019.
[13] X. Gu, Y. Qi, and M. El-Kebir, “DERNA enables Pareto optimal RNA design,” *J. Comput. Biol.*, vol. 31, pp. 179–196, 2024.
[14] R. Nussinov and A. B. Jacobson, “Fast algorithm for predicting the secondary structure of single-stranded RNA,” *Proc. Nat. Acad. Sci. USA*, vol. 77, pp. 6309–6313, 1980.
[15] D. H. Mathews, J. Sabina, M. Zuker, and D. H. Turner, “Expanded sequence dependence of thermodynamic parameters improves prediction of RNA secondary structure,” *J. Mol. Biol.*, vol. 288, pp. 911–940, 1999.
[16] T. H. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.
[17] The UniProt Consortium, “UniProt: The universal protein knowledgebase in 2023,,” *Nucleic Acids Res.*, vol. 51, pp. D523–D531, 2023.
[18] R. Lorenz et al., “ViennaRNA package 2.0,” *Algorithms Mol. Biol.*, vol. 6, 2011, Art. no. 26.
[19] Y. Nakamura, T. Gojobori, and T. Ikemura, “Codon usage tabulated from international DNA sequence databases: Status for the year 2000,,” *Nucleic Acids Res.*, vol. 28, pp. 292–292, 2000.

Yan-Ru Ju received the MS degree from the Institute of Information Systems and Applications, National Tsing Hua University, Taiwan, in 2023. He is currently working as a research assistant with the Institute of Information Science, Academia Sinica, Taiwan. His research interests include design and analysis of algorithms and computational biology.



Long-Shang Cho received the MS degree from the Department of Computer Science, National Tsing Hua University, Taiwan, in 2024. His research interests include design and analysis of algorithms and computational biology.

Chin Lung Lu received the PhD degree in computer science from the National Tsing Hua University, Taiwan, in 1998. He is a professor with the Department of Computer Science, National Tsing Hua University, Taiwan. His research interests include design, analysis and applications of algorithms, mainly focusing on bioinformatics algorithms and graph algorithms.