Improved and Linear-Time Stochastic Sampling of RNA Secondary Structure with Applications to SARS-CoV-2

He Zhang[a,b], **Liang Zhang**[a,b], **Sizhen Li**[b,a], **David H. Mathews**[c,d,e], **and Liang Huang**[b,a,♣]

[a]Baidu Research USA, Sunnyvale, CA; [b]School of Electrical Engineering & Computer Science, Oregon State University, Corvallis, OR; [c]Dept. of Biochemistry & Biophysics; [d]Center for RNA Biology; [e]Dept. of Biostatistics & Computational Biology, University of Rochester Medical Center, Rochester, NY 14642, USA

Many RNAs fold into multiple structures at equilibrium. The classical stochastic sampling algorithm can sample secondary structures according to their probabilities in the Boltzmann ensemble, and is widely used. However, the current sampling algorithm, consisting of a bottom-up partition function phase followed by a top-down sampling phase, suffers from three limitations: (a) the formulation and implementation of the sampling phase are unnecessarily complicated; (b) the sampling phase recalculates many redundant recursions already done during the partition function phase; (c) the partition function runtime scales cubically with the sequence length. These issues prevent it from being used for very long RNAs such as the full genomes of SARS-CoV-2. To address these problems, we first adopt a hypergraph framework under which the sampling algorithm can be greatly simplified. We then present three sampling algorithms under this framework, two of which eliminate redundant work in the sampling phase by caching. Finally, we present LinearSampling, an end-to-end linear-time sampling algorithm that is orders of magnitude faster than the standard one. For instance, LinearSampling is 176× faster (38.9s vs. 1.9h) than Vienna RNAsubopt on the full genome of Ebola virus (18,959 *nt*). More importantly, LinearSampling can scale up to the full genome of SARS-CoV-2, taking only 69.2 seconds on its reference sequence (29,903 *nt*). It finds 23 regions of 15 *nt* with high accessibilities, which can be potentially used for COVID-19 diagnostics and drug design.

See code: https://github.com/LinearFold/LinearSampling

1 Introduction

RNAs are involved in many cellular processes, including expressing genes, guiding RNA modification (8), catalyzing reactions (7) and regulating diseases (15). Many functions of RNAs are highly related to their secondary structures. However, determining the structures using experimental methods, such as X-ray crystallography (40), Nuclear Magnetic Resonance (NMR) (38), or cryo-electron microscopy (20), are expensive, slow and difficult. Therefore, being able to rapidly and accurately predict RNA secondary structures is desired.

Commonly, the minimum free energy (MFE) structure is predicted (29, 42), but these methods do not capture the fact that multiple conformations exist at equilibrium, especially for mRNAs (16, 17, 19, 35). To address this, McCaskill (26) pioneered the partition function-based methods, which account for the ensemble of all possible structures. The partition function can estimate the *base pairing probabilities* $p_{i,j}$ (nucleotide i paired with j), and the *unpaired probabilities* q_i (i is unpaired).

However, the estimated base-pairing and unpaired probabilities $p_{i,j}$'s and q_i's, being marginalized over all possible structures, only provide *compact representations* of the exponentially large ensemble, but can not provide direct and intuitive descriptions (5). First, in many situations, we prefer to see a sample of representative structures according to their Boltzmann probabilities, which is more informative than the marginal probabilities (25). For example, we can use a set of samples to estimate the end-to-end distance of an RNA (16). And more importantly, we often want to predict the probability that a region is completely unpaired, known as the *accessibility* of that region, which plays an important role in siRNA sequence design (3, 11, 19, 35). Accessibility *cannot* be simply computed as the product of the unpaired probabilities for each base in the region because those probabilities are *not* independent.

To alleviate these issues, Ding and Lawrence (5) pioneered the widely-used technique of stochastic sampling, which samples secondary structures according to their probabilities in the ensemble. Their algorithm consists of two phases: the first phase computes the partition function (but not the marginal

♣Corresponding author: liang.huang.sh@gmail.com.

probabilities) in a standard bottom-up fashion, and the second "sampling" phase generates structures in a top-down iterative refinement fashion. This algorithm can estimate ensembles of structures, and predict the accessibility by sampling k structures and counting how many of them have the region of interest completely unpaired. Two popular RNA folding packages, RNAstructure (23) and Vienna RNAfold (18), both implement this algorithm.

However, widely-used as it is, the standard Ding and Lawrence sampling algorithm suffers from three limitations. First, its formulation and implementation are unnecessarily complicated (see Sec. 2.2.2 for details). Secondly, the sampling phase repeatedly recomputes many recursions already performed during the partition function phase, wasting a substantial amount of time especially for large sample sizes.[1] Finally, it relies on the standard partition function calculation that takes $O(n^3)$-runtime, where n is the sequence length. This slowness prevents it from being used for long sequences including the full-genome of SARS-CoV-2.

To alleviate these three issues, we present one solution to each of them. We adopt the hypergraph framework (9, 10), under which the sampling algorithm can be greatly simplified. This framework conveniently formulates the search space of RNA folding, and then sampling can be simplified as recursive stochastic backtracing in the hypergraph.

Under this framework, we present three sampling algorithms. The first one (non-saving) is similar to but much simpler and cleaner than Ding and Lawrence's, while the other two are completely novel. The second one (full-saving) eliminates all redundant calculations by saving all computations from the partition-function phase, and the third one (lazy-saving), being the fastest, is a trade-off between the first two, which only saves computations that are needed during the sampling phase.

Finally, we present LinearSampling, an end-to-end linear-time sampling algorithm that is orders of magnitude faster than the standard algorithm. LinearSampling uses our recently proposed $O(n)$-time LinearPartition algorithm (39) for the partition function calculation, replacing the classical $O(n^3)$-time one, and uses our proposed lazy-saving algorithm for the sampling phase. LinearSampling achieves $176\times$ speedup (38.9s vs. 1.9h) compared to RNAsubopt on the full genome of the Ebola virus (18,959 *nt*).

As the COVID-19 outbreak spreads, it is of great value to find the regions with high accessibilities in SARS-CoV-2, which can be potentially used for diagnostics and drug design. However, there was no tool that can fast sample structures and calculate the accessibilities on such long sequences and consider global, long-range base pairs. LinearSampling takes only 69.2 seconds on the whole-genome of SARS-CoV-2 ($\sim$30,000 *nt*), and finds 23 regions of 15 *nt* with high accessibilities for potential drug design.

2 Sampling Algorithms

We first formulate (in Sec. 2.2.1) the search space of RNA folding using the framework of (directed) hypergraphs (9, 10) which have been used for both the closely related problem of context-free parsing (12) and RNA folding itself (9, 31). This formulation makes it possible to present the various sampling algorithms succinctly (see Sec. 2.2), where sampling can be done in a top-down way that is symmetric to the bottom-up partition function computation. Finally, we present (in Sec. 2.3) our LinearSampling algorithm which is the first sampling algorithm to run in end-to-end linear-time.

2.1 Hypergraph Framework

For an input RNA $\mathbf{x} = x_1 \dots x_n$, we formalize its search space as a **hypergraph** $\langle V(\mathbf{x}), E(\mathbf{x}) \rangle$. Each **node** $v \in V(\mathbf{x})$ corresponds to a subproblem in the search space, such as a span $\mathbf{x}_{i,j}$. Each **hyperedge** $e \in H(\mathbf{x})$ is a pair $\langle node(e), subs(e) \rangle$ which denotes a decomposition of $node(e)$ into a list of children nodes $subs(e) \in V(\mathbf{x})^*$. For example, $\langle \mathbf{x}_{i,j}, [\mathbf{x}_{i,k}, \mathbf{x}_{k+1,j}] \rangle$ divides one span into two smaller ones. For each node v, we define its **incoming hyperedges** to be all decompositions of v:

$$\textsc{InEdges}(v) \triangleq \{e \mid node(e) = v\}$$

We define the **arity** of a hyperedge e, denoted $|e|$, to be the number of its children nodes ($|e| \triangleq$

[1] Our notion of "redundant" is unrelated to the one in "non-redundant sampling" (27), which is a variant to output *unique* samples, while standard sampling can sample the same structure more than once.

$|subs(e)|$). In order to recursively assemble substructures to form the global structure, each hyperedge $e = \langle v, subs \rangle$ is associated with a **combination function** $f(e) : \mathbb{S}^{|e|} \mapsto \mathbb{S}$ that assembles substructures from $subs$ into a structure for v (here $\mathbb{S}$ is the set of dot-bracket strings). Each hyperedge e is associated with an (extra) **energy term** $w(e) \in \mathbb{R}$.

We take the classical Nussinov algorithm (29) as a concrete example, which scores secondary structures by counting the number of base pairs. The nodes are

$$V(\mathbf{x}) = \{\mathbf{x}_{i,j} \mid 1 \le i \le j+1 \le n\}$$

which include both non-empty substrings $\mathbf{x}_{i,j} = x_i ... x_j$ $(i \le j)$ that can be decomposed, and empty spans $\mathbf{x}_{i,i-1}$ $(i = 1...n)$ that are the terminal nodes. Each non-empty span $\mathbf{x}_{i,j}$ can be decomposed in two ways: either base x_j is unpaired (unary) or paired with some x_k $(i \le k < j)$ (binary). Therefore, the incoming hyperedges for $\mathbf{x}_{i,j}$ are

$$\textsc{InEdges}(\mathbf{x}_{i,j}) = \textsc{Unary}(\mathbf{x}_{i,j}) \cup \textsc{Binary}(\mathbf{x}_{i,j})$$

where $\textsc{Unary}(\mathbf{x}_{i,j}) = \{\langle \mathbf{x}_{i,j}, [\mathbf{x}_{i,j-1}] \rangle\}$ contains a single hyperedge with the combination function $f_1(a) =$ "a **.**" that appends an unpaired "**.**" for x_j:

$$\overbrace{\underset{i}{\boxed{\mathbf{x}_{i,j-1}}}\ \underset{j}{\bullet}}^{\mathbf{x}_{i,j}}$$

And the set of binary hyperedges

$$\textsc{Binary}(\mathbf{x}_{i,j}) = \bigcup_{\substack{i \le k < j \\ (x_k, x_j) \text{ pair}}} \{\langle \mathbf{x}_{i,j}, [\mathbf{x}_{i,k-1},\ \mathbf{x}_{k+1,j-1}] \rangle\}$$

contains all bifurcations with (x_k, x_j) paired, dividing $\mathbf{x}_{i,j}$ into two smaller spans $\mathbf{x}_{i,k-1}$ and $\mathbf{x}_{k+1,j-1}$:

$$\overbrace{\underset{i}{\boxed{\mathbf{x}_{i,k-1}}}\ \underset{k}{(}\ \boxed{\mathbf{x}_{k+1,j-1}}\ \underset{j}{)}}^{\mathbf{x}_{i,j}}$$

All these hyperedges share the same combination function $f_2(a, b) =$ "a **(**b**)**" which combines the two substructures along with the new (x_k, x_j) pair. They also share the energy term $w = -1$ (kcal/mol), the stabilizing free energy term for forming a base pair.[2]

[2]Each empty span $\mathbf{x}_{i,i-1}$ has a special nullary hyperedge $\langle \mathbf{x}_{i,i-1}, [\,] \rangle$ with no children, and the associated nullary function f_0 is $f_0(\,) =$ "".

Finally, a special **goal node** $goal(V(\mathbf{x}))$ is identified as the root of the recursion, which in the Nussinov algorithm is the whole sequence $\mathbf{x}_{1,n}$.

This framework can easily extend to other folding algorithms such as Zuker (42) and LinearFold (13), where nodes are *"labeled spans"* such as $C_{i,j}$ for substructures over $\mathbf{x}_{i,j}$ with (x_i, x_j) paired, $M_{i,j}$ for multiloops over $\mathbf{x}_{i,j}$, etc.

2.2 Three Sampling Algorithms

Under the hypergraph framework, we first describe the bottom-up partition function phase (also known as the "inside" or "forward" phase), and then present three algorithms for the top-down sampling phase, i.e., non-saving, full-saving, and lazy-saving. While the first is similar to but cleaner than the standard sampling algorithm, the other two are novel.

2.2.1. The Partition Function Phase. In this bottom-up phase, we first calculate the local partition function $Z(v)$ of each node v (see Fig. 1), summing up the contributions from each incoming hyperedge e (line 7), i.e., $Z(v) = \sum_{e \in \textsc{InEdges}(v)} Z(e)$. This part takes $O(E) = O(n^3)$ time as each hyperedge is traversed once and $O(V) = O(n^2)$ space as we need to store $Z(v)$ for each node v. Note that the hyperedges are by default not saved, and will be recalculated on demand during the sampling phase. If we want to save all hyperedges (for the full-saving algorithm in Sec. 2.2.3) instead, we need $O(n^3)$ space; the time complexity remains $O(n^3)$, but in practice the overhead for saving (line 8) is quite costly and it may run out of memory (see Fig. 6).

2.2.2. Non-Saving Sampling. In the sampling phase, the non-saving sampling algorithm (see Fig. 2) recursively backtraces from the goal node, in a way that is symmetric to the bottom-up partition function phase. When visiting a node v, it tries to sample a hyperedge e from v's incoming hyperedges $\textsc{InEdges}(v)$ according to the probability $Z(e)/Z(v)$. This is done by first generating a random number p between 0 and $Z(v)$, and then gradually recovering each incoming hyperedge e, accumulating its $Z(e)$ to a running sum s, until s exceeds p, at which point that current hyperedge e is chosen as sampled. Note that this algorithm in general does *not* need to recover

```
function PartFunc(V, saving)                    ▷ calculate partition function
  for each v in BottomUpOrder(V) do             ▷ traverse nodes
    Z(v) ← SumEdges(v, saving)                  ▷ partition function for v
function SumEdges(v, saving)                    ▷ traverse v's hyperedges
  s ← 0
  for each e = ⟨v, subs⟩ in InEdges(v) do
    s += exp(−w(e)/RT) · ∏_{u∈subs} Z(u)        ▷ accumulate Z(e) to Z(v)
    if saving then S(v) ← S(v) ∪ {(e, s)}       ▷ save hyperedge
  return s
```

Fig. 1. The bottom-up phase to calculate the partition function.

```
function Sample(v)                              ▷ recover just enough hyperedges
  p ← random(0, Z(v)); s ← 0
  for each e = ⟨v, subs⟩ in InEdges(v) do
    s += exp(−w(e)/RT) · ∏_{u∈subs} Z(u)
    if s > p then return f(e)(Sample(u) for u in subs)
function Main(x, k)                             ▷ k is the sample size
  PartFunc(V(x), False)                         ▷ do not save hyperedges
  for i = 1...k do Sample(goal(V(x)))
```

Fig. 2. The non-saving sampling algorithm. This version is similar to Ding and Lawrence (5) but much simpler and cleaner thanks to the bottom-up↔top-down symmetry.

```
function SampleSave(v)
  sample e = ⟨v, subs⟩ from S(v)                ▷ binary search
  return f(e)(SampleSave(u) for u in subs)
function MainSave(x, k)
  PartFunc(V(x), True)                          ▷ save all hyperedges
  for i = 1...k do SampleSave(goal(V(x)))
```

Fig. 3. The full-saving sampling algorithm.

all incoming hyperedges of v, though in the worst case it would. It then backtraces recursively to the corresponding subnode(s) of hyperedge e.

Now let us analyze the time complexity to generate each sample. First of all, it visits $O(n)$ nodes to generate one sample as there are $O(n)$ nodes in each derivation (i.e., the recursion tree). On each node $v = \mathbf{x}_{i,j}$, it needs to recover $O(|j-i|)$ hyperedges, so the total number of hypereges recovered depends on how balanced the derivation is, similar to quicksort. In the worst case (when the derivation is extremely unbalanced like a chain), it recovers $O(n^2)$ hyperedges, and in the best case (when the derivation is mostly balanced, i.e., bifurcations near the middle), it only recovers $O(n \log n)$ hyperedges. So the time to generate k samples is $O(kn^2)$ (worst-case) or $O(kn \log n)$ (best-case).[3] Our experiments in Sec. 3 (Fig. SI 7) show that, like in quick sort, the sampled derivations are mostly balanced as the depth of derivation scales $O(\log n)$ in practice, thus the average case behavior is essentially best case.[4]

This version is closest to the original Ding and Lawrence (5) algorithm, but simpler and cleaner. Our key idea is to exploit the structural symmetry between the bottom-up and sampling phases, and unify them under the general hypergraph framework. By contrast, Ding and Lawrence do *not* exploit this symmetry, and instead rely on different recurrences in the sampling phase that iteratively samples the leftmost external pair in an external span and the rightmost pair in a multiloop (see Fig. 1 of their paper). Their formulation results in unnecessarily complicated implementations (see Vienna RNAsubopt for an example).[5] We are the first to formulate general sampling (Nussinov, Zuker, LinearFold, etc.) under a unified framework that exploits symmetry.[6]

2.2.3. Full-Saving Sampling. It is obvious that the non-saving version wastes time recovering hyperedges during the sampling phase. First, due to the symmetry, all hyperedges recovered in the sampling phase have already been traversed during the inside phase. To make things worse, many hyperedges are recovered multiple times across different samples because whenever a node is (re-)visited, its hyperedges need to be re-recovered. This situation worsens with the sample size k. More formally, we define

$$\alpha_k \triangleq \frac{\text{\# of unique nodes visited in sampling}}{\text{\# of all node visits in sampling}} \quad [1]$$

to be the "unique visit ratio" among k samples, and we will see in Fig. 8A that this ratio is extremely small, quickly approaching 0% as k increases, meaning most node visits are repeated visits. This begs the question: why don't we save all hyperedges during the inside phase, so that no hyperedge needs to

[3] For each sample, worst-case: $T(n) = T(n-1) + O(n) = O(n^2)$, and best-case: $T(n) = 2T(n/2) + O(n) = O(n \log n)$.

[4] Ponty (30) applies the "Boustrophedon" method to reduce the worst-case time also to $O(n \log n)$. Our experiments (Fig. SI 7) show that it does further improve the runtime, but only slightly.

[5] We point out that RNAstructure (24)'s sampling is similar to our non-saving version except for being non-recursive, i.e., iterative.

[6] Ponty (30) analyzes the special case of sampling under the Nussinov model by exploiting the symmetry (though it could have generalized to other systems), and he implemented on the simplified Nussinov-Jacobson (29) rather than the full Turner model (22) as in our work.

partition function	sampling algorithm	partition function time	partition function space	sample time (worst-case)	sample time (best-case)	sample space (worst-case)
exact	**non-saving**	$O(n^3)$	$O(n^2)$	$O(kn^2)$	$O(kn \log n)$	$O(1)$
	full-saving		$O(n^3)$	$O(kn \log n)$	$O(kn)$	
	lazy-saving		$O(n^2)$	$O(\alpha_k kn^2 + kn \log n)$	$O(\alpha_k kn \log n + kn)$	$O(\beta_k n^3)$
linear	**non-saving**	$O(nb^2)$	$O(nb)$	$O(knb)$	$O(kn \log b)$	$O(1)$
	full-saving		$O(nb^2)$	$O(kn \log b)$	$O(kn)$	
	lazy-saving		$O(nb)$	$O(\alpha_k knb + kn \log b)$	$O(\alpha_k kn \log b + kn)$	$O(\beta_k nb^2)$

Table 1. Complexities for exact and linear-time partition function calculations coupled with three sampling algorithms (n is the sequence length, k is sample size, α_k and β_k are the "unique visit ratio" (Eq. 1) and "visited ratio" (Eq. 2), resp., and b is beam size). The lazy-saving runtimes are hybrids between the non- and full-saving ones, i.e., $T^{\text{S}}_{\text{lazy}} = \alpha_k T^{\text{S}}_{\text{non}} + T^{\text{S}}_{\text{full}}$ (Eq. 3).

be recovered during the sampling phase? To address this we present the second, full-saving version (see Figs. 2– 3), which saves for each node v the contributions $Z(e)$ of each hyperedge e to the local partition function $Z(v)$, once and for all. Then the sampling phase is easier, only requiring sampling a hyperedge e according to its relative contribution (or "weight") to v, i.e., $Z(e)/Z(v)$ (line 2 in Fig. 3). Actually, modern programming languages such as C++ and Python provide tools for sampling from a weighted list, which is implemented via a binary search in the sorted array of cumulative weights (which is why line 8 in Fig. 1 saves the running sum rather than individual contribution $Z(e)$). This takes only $O(\log n)$ time for each v as $|\textsc{InEdges}(v)| = O(n)$ (consider all bifurcations). Therefore, the worst-case complexity for generating k samples is $O(kn \log n)$ and the best-case is $O(kn)$.[7]

2.2.4. Lazy-Saving Sampling. Though the full-saving version avoids all re-calculations, it costs too much more space ($O(n^3)$ vs. $O(n^2)$) and significantly more time in practice for saving the whole hypergraph. Actually, the vast majority of nodes are *never* visited during the sampling phase even for large sample size. To quantify this, we define

$$\beta_k \triangleq \frac{\text{\# of unique nodes visited in sampling}}{\text{\# of all nodes in hypergraph}} \quad [2]$$

to be the "visited ratio". Our experiments in Fig. 8B show that only $< 0.5\%$ of all nodes in the hypergraph are ever visited for 20,000 samples of a 3,048 *nt* sequence using Vienna RNAsubopt, so most of the saving is indeed wasted. Based on this, we present our third version, lazy-saving, which is a hybrid between non-saving and full-saving versions (see Fig. 4). By "lazy" we mean only recovering and saving a node v's hyperedges when needed, i.e., the first time v is visited during sampling phase. In this way each hyperedge is recovered at most once, and most are not recovered at all. This version balances between space and time, and is the fastest among the three versions in most settings in practice.

The complexity analysis of lazy-saving is also a hybrid between the non- and full-saving versions, combined together using the α_k and β_k ratios. We note that lazy saving sampling time consists of two parts: (a) the hyperedge-recovering (and saving) work, and (b) the sampling work after relevant hyperedges are recovered. Part (a) resembles non-saving sampling, but with a ratio of α_k, because most node visits are repeated ones, and once a node is visited for the first time in sampling, its hyperedges are recovered and saved, and all future visits to this node will be like full saving version. Part (b) is identical to the full-saving version (in both cases, all needed hyperedges are already saved). Therefore, we have the following relations among the time complexities

```
function SampleLazy(v, visited)
  if v ∉ visited then                    ▷ first visit to v in sampling phase
    SumEdges(v, True)                    ▷ recover and save v's hyperedges
    visited ← visited ∪ {v}
  else                                   ▷ v's hyperedges already recovered
    sample a hyperedge e = ⟨v, subs⟩ from S(v)
    return f(e)(SampleLazy(u, visited) for u in subs)
function MainLazy(x, k)
  PartFunc(V(x), False)                  ▷ do not save during PartFunc
  visited ← ∅
  for i = 1...k do SampleLazy(goal(V(x)), visited)
```

Fig. 4. The lazy-saving sampling algorithm.

[7] For each sample, worst-case $T(n) = T(n-1) + \log n = O(n \log n)$; best-case $T(n) = 2T(n/2) + \log n = O(n)$, similar to "heapify" (4).

for the sampling phase of these three versions:

$$T^{\mathrm{S}}_{\mathrm{lazy}}(n,k) = \alpha_k T^{\mathrm{S}}_{\mathrm{non}}(n,k) + T^{\mathrm{S}}_{\mathrm{full}}(n,k) \qquad [3]$$

This holds for both the worst and best-case scenarios in Tab. 1. The space complexity is easier: lazy saving saves only a fraction (β_k) of all nodes in the hypergraph, thus $O(\beta_k n^3)$. See Tab. 1 for summary.

2.3 LinearSampling = LinearPartition + Lazy-Saving Sampling

Lazy-saving sampling is the most efficient among all three methods presented above, but the biggest bottleneck remains the $O(n^3)$-time partition function computation, which prevents it from scaling to full-length viral genomes such as SARS-CoV-2. To address this problem, we replace our recently proposed linear-time approximate algorithm, LinearPartition (39), to replace the standard cubic-time one. It can be followed by any one of the three sampling algorithms (non-, full-, and lazy-saving) for the sampling phase, and in particular, we name the one with lazy-saving the LinearSampling algorithm as it is the fastest among all combinations.

Fig. 5 describes a simplified pseudocode using the Nussinov-Jacobson energy model. Inspired by LinearPartition, we employ beam search to prune out nodes with small partition function (line 11) during the inside phase. So at each position j, only the top b promising nodes "survive" (i.e., $O(nb)$ nodes survive in total). Here the beam size b is a user-specified hyperparameter, and the default $b = 100$ is found to be more accurate for structure prediction than exact search (39). The partition function runtime is reduced to $O(nb^2)$ (there are only $O(b)$ hyperedges per node) and the space complexity is reduced to $O(nb)$, both of which are linear against sequence length n. The sampling time is also linear regardless of non-, full-, or lazy-saving. The binary search time to sample a saved hyperedge reduces from $O(\log n)$ to $O(\log b)$ since at most b hyperedges are saved for each node (thanks to beam search). Then, following Eq. 3, we can derive the complexities in Table 1. In particular, the LinearSampling algorithm (the last line in the table) has an end-to-end runtime of $O(nb^2 + \alpha_k kn \log b + kn)$ and uses $O(nb + \beta_k nb^2)$ space in total, both of which scales linearly in n.

```
1:  function LINEARPARTFUNC(x, b, saving)          ▷ b: beam size
2:    n ← length of x
3:    Z ← hash()                                   ▷ hash table: from x_{i,j} to Z(x_{i,j})
4:    Z(x_{j,j-1}) ← 1 for all j in 1...n          ▷ base cases (empty spans)
5:    for j = 1...n do
6:      for each i such that x_{i,j-1} in Z do     ▷ O(b) iterations
7:        UPDATE(⟨x_{i,j}, [x_{i,j-1}]⟩, saving)   ▷ unary hyperedge
8:        if (x_{i-1}, x_j) match then             ▷ x_j pairs with x_{i-1}
9:          for each k such that x_{k,i-2} in Z do ▷ O(b) iterations
10:           UPDATE(⟨x_{i,j+1}, [x_{k,i-2}, x_{i,j-1}]⟩, saving)   ▷ binary
11:     BEAMPRUNE(Z, j, b)   ▷ choose top b out of Z(x_{i,j}) for all i
12: function UPDATE(e, saving)
13:   ⟨v, subs⟩ = e
14:   Z(v) += exp(-w(e)/RT) · ∏_{u∈subs} Z(u)       ▷ add Z(e) to Z(v)
15:   if saving then S(v) ← S(v) ∪ {(e, Z(v))}
16: function INEDGES(x_{i,j})
17:   if j = i-1 then return {⟨x_{i,i-1}, []⟩}     ▷ base case (nullary)
18:   if x_{i,j-1} in Z then yield ⟨x_{i,j}, [x_{i,j-1}]⟩
19:   for k = i...j-1 s.t. x_{k+1,j-1} in Z do
20:     if (x_k, x_j) match and x_{i,k-1} in Z then
21:       yield ⟨x_{i,j}, [x_{i,k-1}, x_{k+1,j-1}]⟩
```

Fig. 5. Part of the (simplified) pseudocode of linear-time sampling algorithms on the Nussinov-Jacobson energy model, which can be called by the SAMPLE (Fig. 2), SAMPLESAVE (Fig. 3), and SAMPLELAZY (Fig. 4) functions, by replacing PARTFUNC in those functions with LINEARPARTFUNC here. The INEDGES function is used in the SUMEDGES function in the non-saving and lazy-saving versions. See Fig. SI 1 for the pseudocode of beam pruning (line 11). The actual algorithm using the Turner model is available on GitHub.

3 Results

3.1 Efficiency and Scalability

We benchmark the runtime and memory usage on 26 sequences sampled from RNAcentral (33). We evenly split the range from 0 to 8,000 into 52 bins by log-scale, and randomly select at most one sequence in each bin; within 100 *nt* only one sequence is chosen. We refer this dataset as the RNAcentral dataset in the paper. We use a Linux machine (CentOS 7.5.1804) with 3.40 GHz Intel Xeon E3-1231 v3 CPU and 16 GB memory, and gcc 4.8.5.

3.1.1. Comparing Non-, Full-, and Lazy-Saving Samplings on Exact Partition Function. Fig. 6 shows the performance of different versions under the exact partition function calculation, using Vienna RNAsubopt (18) as a baseline, and our own exact partition function (basically, setting $b = +\infty$) which is faster than Vienna with identical results. Regarding the end-to-end runtime, full-saving is the slowest

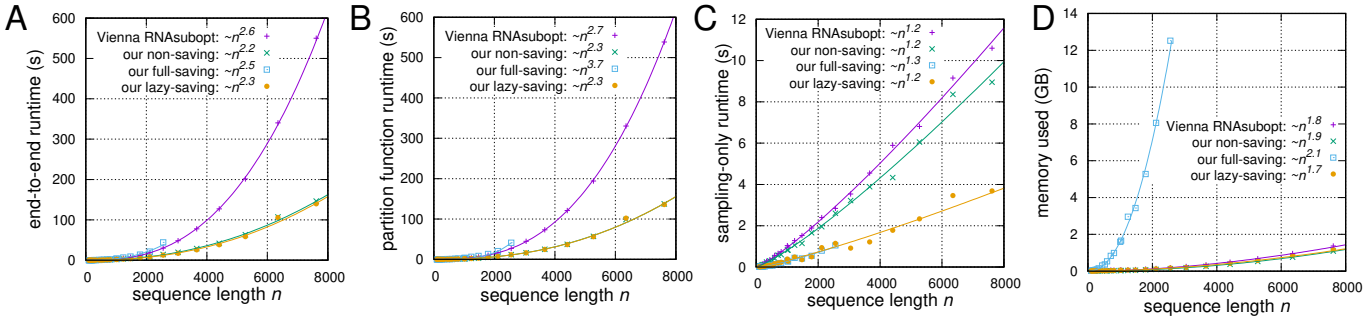

Fig. 6. Runtime and memory comparisons against sequence length n on RNAcentral dataset, using exact partition function ($k = 10,000$ samples). Our exact partition function (non- and lazy-saving) is faster than Vienna (with identical outputs) (B). Our non-saving sampling is slightly faster than Vienna (C), full-saving runs out of memory on 3,048 *nt*, and lazy-saving is the fastest.

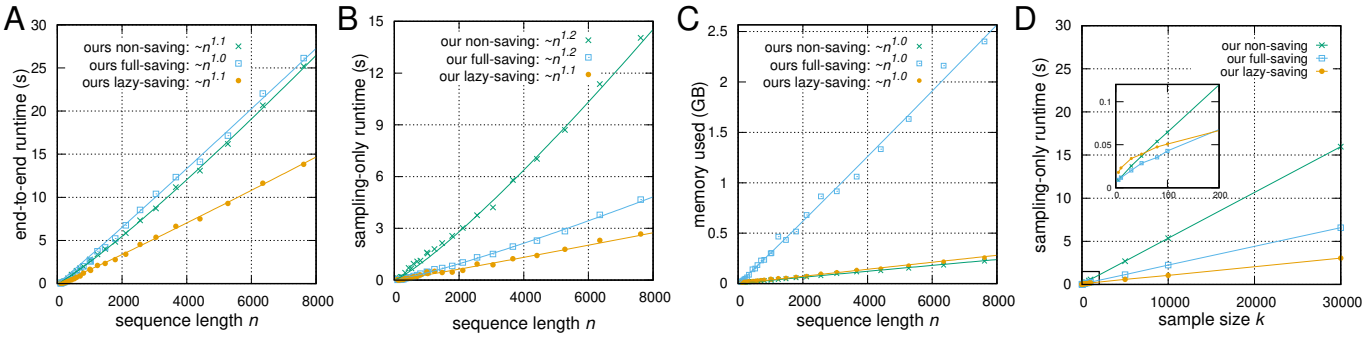

Fig. 7. Runtime and memory usage comparisons using linear partition function. **A**-**C**: comparison against sequence length n with sample size 10,000. **D**: comparison against sample size k on a 3,048 *nt* sequence in the RNAcentral dataset.

since it spends much time on hyperedges saving, and it runs out of memory on a 3,048 *nt* sequence. Lazy-saving is slightly faster than non-saving, and they are both faster than RNAsubopt. The partition function runtime is close to end-to-end, showing that the partition function computation takes the majority of the total runtime. Although all are fast during sampling, lazy-saving are more than $2.5\times$ faster than non-saving and RNAsubopt. Regarding memory usage, full-saving uses much more memory, while the other three are close.

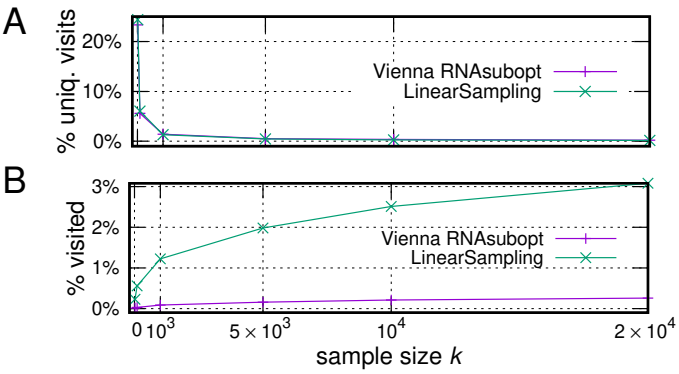

Fig. 8. In practice most node visits are repeated, and only a small portion of all nodes are visited. **A**: unique visit ratio α_k. **B**: visited ratio β_k. Here n=3,048 *nt*. Fig. SI 3 demonstrates the trend with sequence length, and Fig. SI 4 demonstrates that most of the visits are concentrated on a few nodes.

3.1.2. Comparing Non-, Full-, and Lazy-Saving Samplings on Linear Partition Function. For the end-to-end runtime (Fig. 7A), lazy-saving is the fastest, followed by non-saving, and then by full-saving. For the sampling-only runtime (Fig. 7B), clearly non-saving is the slowest, while lazy-saving is the fastest; full-saving is slower than lazy-saving due to the high memory usage pressure, and we observe that full-saving is slightly faster than lazy-saving on a 100 *nt* sequence (0.05 secs vs. 0.01 secs). Fig. 7C confirms that full-saving requires the most memory, while non-saving and lazy-saving reduce 90% of memory usage.

Fig. 7D compares sampling-only runtime against sample size k on a 3,048 *nt* sequence. Clearly, lazy-saving is the fastest for most cases. We observe that non-saving is slightly faster than lazy-saving when $k < 50$, but is the slowest otherwise. Full-saving is slightly faster than lazy-saving when $k < 200$, and in the middle of non-saving and lazy-saving

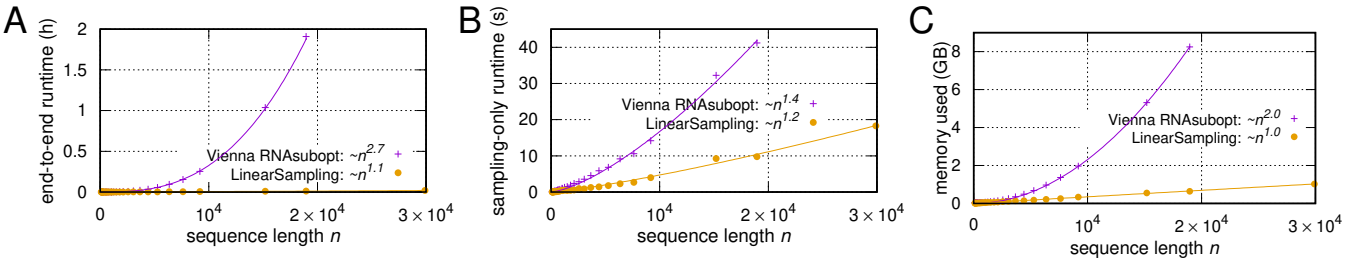

Fig. 9. Runtime and memory usage comparisons between RNAsubopt and LinearSampling on the RNAcentral dataset and viral sequences. The sample size is 10,000. Note that LinearSampling can scale to the full genome of SARS-CoV-2 (∼30,000 *nt*).

above that. Fig. SI 5 shows more comparisons on SARS-CoV-2 and the 3,048 *nt* sequence.

We also illustrate why lazy-saving is the fastest in practice. Fig. 8A presents the unique visit ratio α_k is less than 5% when $k > 1,000$ for both RNAsubopt and LinearSampling, confirming that lazy-saving is able to avoid a large number of re-calculations during the sampling phase. On the other hand, the visited ratio β_k (Fig. 8B) is always smaller than 0.5% and 3% for RNAsubopt and LinearSampling, resp, and grows slower and slower as the sampling size is increasing, showing that saving all hyperedges (i.e., full-saving) is not ideal.

It is clear that the lazy-saving version is the fastest among the three versions, so we use it as default in LinearSampling, and all the results in the rest of the paper are based on lazy-saving.

3.1.3. Comparing Vienna RNAsubopt and LinearSampling. Next, we compare the efficiency and scalability between LinearSampling and RNAsubopt on full-length viral genomes, so we extend our benchmark dataset by adding four viral sequences, HIV (9,181 *nt*), RSV (15,191 *nt*), Ebola (18,959 *nt*) and SARS-CoV-2 (29,903 *nt*).

Regarding end-to-end runtime, LinearSampling scales almost linearly against sequence length, and is much faster than RNAsubopt. LinearSampling is 176× faster (38.9s vs. 1.9h) than RNAsubopt on the full genome of Ebola virus (18,959 *nt*), and can finish the full-length of SARS-CoV-2 in 69.2s; while RNAsubopt runs out of memory on SARS-CoV-2. Regarding sampling-only runtime, LinearSampling is more than 3× faster. Fig. 9C confirms that the memory usage of LinearSampling is linear, but RNAsubopt requires $O(n^2)$ memory. LinearSampling uses less than 1 GB memory for Ebola sequence, while RNAsubopt uses more than 8 GB.

We also observe that, similar to RNAfold, the RNAsubopt tool sometimes overflows on long sequences during the partition function calculation (39), making it less reliable for long sequences.

3.2 Quality of the Samples

We use the ArchiveII dataset (24, 34), which contains a set of sequences with well-determined structures, to investigate the quality of the samples. We follow the preprocessing steps of a previous study (39), and obtain a subset of 2,859 sequences distributed in 9 families.

3.2.1. Approximation Quality to Base Pairing Probabilities. To evaluate if the sampling structures approximate to the ensemble distribution, Ding and Lawrence (5) investigated the frequency of the MFE structure appeared in the samples, and checked if it matches with the Boltzmann distribution. However, this only works for short sequences because the frequency of the MFE structure is extremely small for long sequences, e.g., 2.23×10^{-32} for *E. coli* 23S rRNA. Alternatively, we investigate the root-mean-square deviation (RMSD) between the base pairing probability matrices $p(S)$, which is derived from the sample set S, and p', which is generated by Vienna RNAfold or LinearPartition. Note that RMSD is averaged on all possible Watson-Crick and G-U pairs on the sequence (41).

Fig. 10A shows three curves of average RMSD against sample size on the ArchiveII dataset. The green curve illustrates the RMSD between LinearSampling and RNAfold. Although LinearSampling approximates the partition function based on LinearPartition, which introduces a small change in the base pairing probabilities (39), the RMSD is only 0.015 with sample size 10, and drops down to 0.005 quickly with sample size 5,000. Regarding

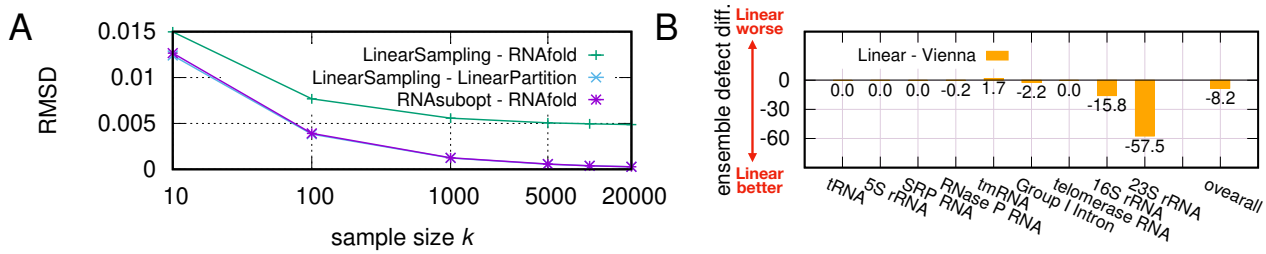

Fig. 10. LinearSampling matches with the ensemble distribution, and is better correlated to the ground truth structure. **A**: the root-mean-square deviation (RMSD) against sample size. The RMSD is averaged within each family, and then averaged on all families. **B**: the ensemble defect difference of each family (overall is averaged by families).

the RMSD between LinearSampling and LinearPartition (the blue curve), and RNAsubopt and RNAfold (the purple curve). we observe that they are almost identical, suggesting LinearSampling can generate structures strictly matching with the ensemble distribution as well as RNAsubopt.

3.2.2. Correlation with the Ground Truth Structure.

We investigate the sampled structure's correlation with the ground truth structure using "ensemble defect" (37), the expected number of incorrectly predicted nucleotides over the ensemble. It is defined:

$$\Phi(S, \mathbf{y}^*) = \frac{1}{|S|} \sum_{\mathbf{y} \in S} d(\mathbf{y}, \mathbf{y}^*)$$
$$= |\mathbf{y}^*| - 2 \sum_{(i,j) \in \text{pairs}(\mathbf{y}^*)} p_{i,j}(S) - \sum_{j \in \text{unpaired}(\mathbf{y}^*)} q_j(S)$$

where $\mathbf{y}^*$ is the ground truth structure, and $d(\mathbf{y}, \mathbf{y}^*)$ is the distance between $\mathbf{y}$ and $\mathbf{y}^*$, defined as the number of incorrectly predicted nucleotides in $\mathbf{y}$. And $q_j(S)$ is the probability of j being unpaired in the sample S, i.e., $q_j(S) = 1 - \sum p_{i,j}(S)$.

Fig. 10B shows the ensemble defect difference between LinearSampling and RNAsubopt on each family (ordered in their average sequence length, from the shortest to the longest) and overall. Note that better correlation to the ground truth structures requires lower ensemble defect. For short families, the difference is either 0 or close to 0, indicating that the sampling qualities of RNAsubopt and LinearSampling are similar on these families. But on 16S and 23S rRNAs, LinearSampling has lower ensemble defect, showing it performs better on longer sequences. The only family that LinearSampling performs worse is tmRNA. In Fig. SI 6 we also present the comparison of RNAsubopt local mode, with base pair length limitations of 70 and 150.[8] It is obvious that the local sampling has much higher (worse) ensemble defect on 23S rRNA, since it ignores all base pairs beyond the span size.

	E. coli	*T. thermophilus*	*S. aureus*	*B. subtilis*	*H. pylori*	overall
RNAsubopt	0.1317	0.1289	**0.1359**	0.1299	**0.1762**	0.1405
ours	**0.1185**	**0.1247**	0.1388	**0.1248**	0.1786	**0.1371**
diff.	-0.0132	-0.0042	0.0029	-0.0051	0.0024	-0.0034

Table 2. Accessibility defect comparison on 23S rRNA family with sample size of 10,000. Ours is LinearSampling.

An important application of the sampling algorithm is to calculate a region's accessibility to base pairing.[9] Therefore, we calculate accessibilities of window size 4 (5) from structures generated by the two systems, and evaluate based on the ground truth structures. We denote the measurement as *accessibility defect*, $D(S, \mathbf{y}^*)$, which is defined as:

$$D(S, \mathbf{y}^*) = \frac{1}{|\mathbf{y}^*| - 3} \sum_{i=1}^{|\mathbf{y}^*|-3} \Big| \text{acc}(\{\mathbf{y}^*\}, i) - \text{acc}(S, i) \Big|$$
$$\text{acc}(S, i) = \frac{1}{|S|} \sum_{\mathbf{y} \in S} \mathbb{1}[\mathbf{y}_{i,i+3} = \text{"....."}]$$

where $\text{acc}(S, i)$ is the accessibility of region $[i, i+3]$.

Table 2 shows the accessibility defect comparison on the ArchiveII 23S rRNA family. We observe that LinearSampling outperforms RNAsubopt on three sequences and the overall result. In Table SI 1, we also present the results of RNAsubopt local mode (span 70 and 150), which are worse than LinearSampling on all 5 sequences. It is surprised that the local

[8] RNAsubopt local mode does not have a default span size; we choose 70 following the default setting in RNAplfold (1), and 150 since it is the largest default limit in the local folding literature and software.

[9] There exist other methods to estimate accessibility, including (a) constrained partition function (forcing each region of interest to be fully unpaired, and compute the fraction of the resulting constrained partition function over the global one), and (b) direct computation (2, 28). But they all run in at least $O(n^3)$ time.

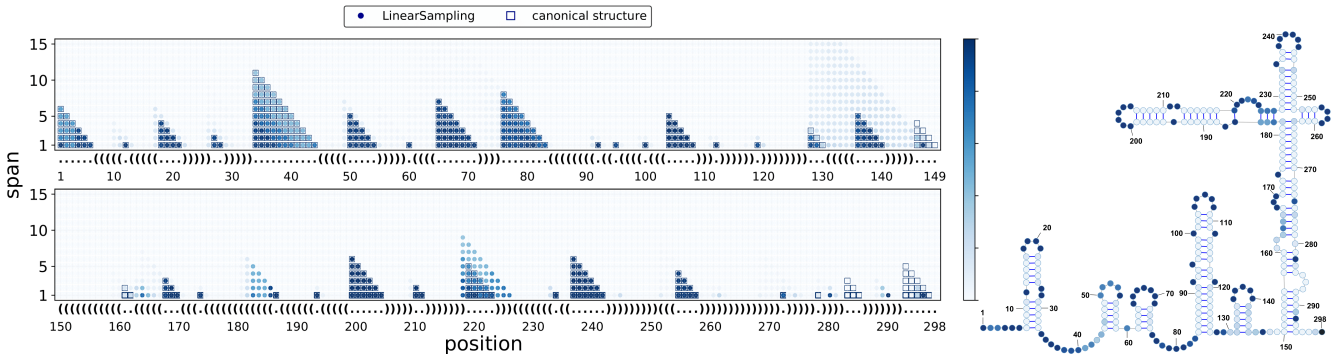

Fig. 11. The accessibilities derived from LinearSampling correlate well with the unpaired region in the canonical structure of SARS-CoV-2 5'-UTR (21). Note that the full sequence was used for the accessibility calculation, but we only illustrate the 5'-UTR region in the figure because its structure is well-established. **A**: accessibilities predicted by LinearSampling with window sizes from 1 to 15. Each prediction is presented with a solid circle, where the darkness correlates to the accessibility value. The accessible regions in the canonical structure are annotated in boxes, and the canonical structure is also shown in the dot-bracket format on the x-axis. **B**: SARS-CoV-2 5'-UTR conanical structure colored with unpaired probabilities generated from LinearSampling sampled structures. The total sample size is 10,000 for both **A** and **B**.

sampling of RNAsubopt are even slower than its global mode. The comparisons on other families are presented in Fig. SI 2, which shows that LinearSampling is better than RNAsubopt on most families, and better than its local sampling on all families.

It is worth noting that the better results of ensemble defect and accessibility defect are inherited from LinearPartition, which correlates better to the ground truth structures (39).

3.3 Applications to SARS-CoV-2

The COVID-19 pandemic has swept the world in 2020, and is likely to be a threaten of global health for a long time. Therefore, it is of great value to find the regions with high accessibilities in SARS-CoV-2, which can be potentially used for COVID-19 diagnostics and drug design. But since SARS-CoV-2 is as long as 30,000 *nt*, existing computational tools are unable to be applied to its full-length genome. Now with significant improvement on sampling efficiency and scalability, LinearSampling is able to fast sample structures for the whole-genome of SARS-CoV-2, and predict its accessible regions.

We run LinearSampling on NC_0405512.2, the reference sequence of SARS-CoV-2 (36). First, we check if the accessibilities predicted by LinearSampling match with the well-established structures, e.g., the 5'-UTR region which has conserved structures and plays a critical role in viral genome replication (21). We sample 10,000 structures, calculate the accessibilities and compare them with the canonical structure. Fig. 11A shows the correlations of predicted accessibilities (solid circles) and the accessible regions in the canonical structure (hollow boxes), in window sizes of 1 to 15 nucleotides. For instance, the dark circle at position 50 and window size 5, representing a highly accessible region [50, 54] predicted by LinearSampling, is surrounded by a box, which indicates that the prediction is supported by the canonical structure. In general, the regions with high accessibilities derived from LinearSampling correlate well to the unpaired regions in the canonical structure. To better illustrate the correlation, we also color the accessibilities of window size 1 (i.e., unpaired probabilities, see the bottom row in Fig. 11A) on the 5'-UTR structure in Fig. 11B.

Secondly, we aim to obtain potentially accessible regions. A previous study (32) locates conserved unstructured regions of SARS-CoV-2 by scanning the reference sequence with windows of 120 *nt*, sliding by 40 *nt*, and then calculating base pairing probabilities using CONTRAfold (6) for these fragments. In total, 75 accessible regions with 15 or more nucleotides are claimed, where each base has the average unpaired probability of at least 0.6. However, this method has two flaws: (1) it is *not* correct principally to evaluate accessibility based on unpaired probabilities due to their mutual dependency; and (2) it neglects long-range base pairs and has to approximate the accessibilities based on local structures.

Instead, we measure the accessibilities based on samples generated by LinearSampling, setting the window size to be 15 following Rangan et al. (32).

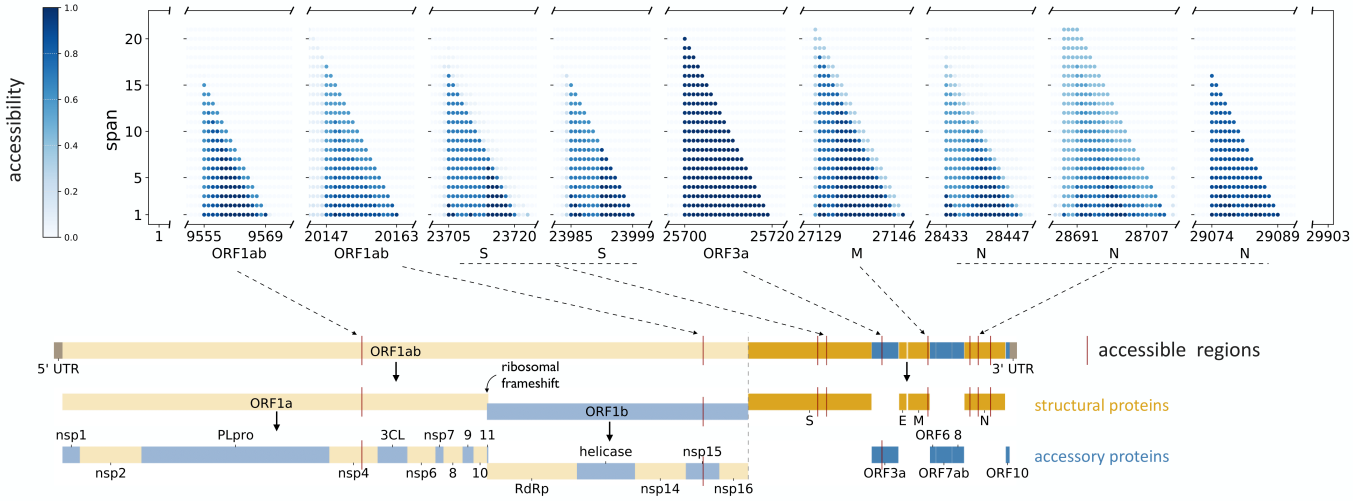

Fig. 12. LinearSampling predicts accessible regions in the SARS-CoV-2 full genome. Note that ORF1ab can be further divided.

We only show the fragments whose accessibilities are larger than 0.5, i.e., they are more likely to be opening than closing. We list all 23 regions found by LinearSampling in Table SI 3. Some of the regions are overlapped, resulting in a total of 9 separate accessible regions, which are illustrated in Fig. 12. Among the 9 regions, two are in ORF1ab, one in ORF3a, one in the M gene, three in the N gene, and two in the S (spike) gene, whose proteins can recognize and bind with receptor (14).

4 Discussion

We focus on simplifying and accelerating the stochastic sampling algorithm for a given RNA sequence. Algorithmically, we present a hypergraph framework under which the classical sampling algorithm can be greatly simplified. We further elaborate this sampling framework in three versions: the non-saving that recovers the hyperedges in a top-down way, the full-saving that saves all hyperedges a priori and avoids re-computing for sampling, and the lazy-saving that only recovers and saves hyperedges on demand. Then we present LinearSampling which combines LinearPartition and lazy-saving sampling.

LinearSampling is the first algorithm to run in end-to-end linear-time without imposing constraints on the base pair distance, and is orders of magnitude faster than the widely-used Vienna RNAsubopt. We confirmed: (1) LinearSampling takes linear runtime and can scale up to long RNA sequence; (2) it approximates well to the ensemble distribution; (3) it correlates better to the ground truth structures; and (4) it can be applied to SARS-CoV-2 for discovering regions with high accessibilities, which are the potential targets for diagnostics and drug design.

References

1. S. H. Bernhart, I. L. Hofacker, and P. F. Stadler. Local RNA base pairing probabilities in large sequences. *Bioinformatics*, 22(5):614–615, 2006.
2. S. H. Bernhart, U. Mückstein, and I. L. Hofacker. Rna accessibility in cubic time. *Algorithms for Molecular Biology*, 6(1):1–7, 2011.
3. E. A. Bohula, A. J. Salisbury, M. Sohail, M. P. Playford, J. Riedemann, E. M. Southern, and V. M. Macaulay. The efficacy of small interfering RNAs targeted to the type 1 insulin-like growth factor receptor (igf1r) is influenced by secondary structure in the igf1r transcript. *Journal of Biological chemistry*, 278(18):15991–15997, 2003.
4. T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms, 3rd edi.* MIT Press, 2009.
5. Y. Ding and C. E. Lawrence. A statistical sampling algorithm for RNA secondary. *Nucleic Acids Research*, 31 (24):7280–7301, 2003.
6. C. Do, D. Woods, and S. Batzoglou. CONTRAfold: RNA secondary structure prediction without physics-based models. *Bioinformatics*, 22(14):e90–e98, 2006.
7. J. A. Doudna and T. R. Cech. The chemical repertoire of natural ribozymes. *Nature*, 418(6894):222–228, 2002.
8. S. R. Eddy. Non-coding RNA genes and the modern RNA world. *Nature Reviews Genetics*, 2(12):919–929, 2001.
9. A. Finkelstein and M. Roytberg. Computation of biopolymers: a general approach to different problems. *BioSystems*, 30(1-3):1–19, 1993.
10. G. Gallo, G. Longo, and S. Pallottino. Directed hypergraphs and applications. *Discrete Applied Mathematics*, 42(2):177–201, 1993.
11. M. Hammell, D. Long, L. Zhang, A. Lee, C. S. Carmack, M. Han, Y. Ding, and V. Ambros. mirWIP: microRNA target prediction based on microRNA-containing

ribonucleoprotein-enriched transcripts. *Nature Methods*, 5(9):813–819, 2008.
12. L. Huang and D. Chiang. Better k-best parsing. *Proc. Int'l Workshop on Parsing Tech. (IWPT)*, pages 53–64, 2005.
13. L. Huang, H. Zhang, D. Deng, K. Zhao, K. Liu, D. Hendrix, and D. Mathews. LinearFold: linear-time approximate RNA folding by 5'-to-3' dynamic programming and beam search. *Bioinformatics*, 35(14):i295–i304, 2019.
14. Y. Huang, C. Yang, X.-f. Xu, W. Xu, and S.-w. Liu. Structural and functional properties of SARS-CoV-2 spike protein: potential antivirus drug development for COVID-19. *Acta Pharmacologica Sinica*, 41(9):1141–1149, 2020.
15. J. T. Y. Kung, D. Colognori, and J. T. Lee. Long noncoding RNAs: Past, present, and future. *Genetics*, 193(3): 651–669, 2013.
16. W.-J. C. Lai, M. Kayedkhordeh, E. V. Cornell, E. Farah, S. Bellaousov, R. Rietmeijer, E. Salsi, D. H. Mathews, and D. N. Ermolenko. mRNAs and lncRNAs intrinsically form secondary structures with short end-to-end distances. *Nature Communications*, 9(1):1–11, 2018.
17. D. Long et al. Potent effect of target structure on microRNA function. *Nature Structural & Molecular Biology*, 14(4):287–294, 2007.
18. R. Lorenz et al. ViennaRNA package 2.0. *Algorithms for Molecular Biology*, 6(1):1, 2011.
19. Z. J. Lu and D. H. Mathews. Efficient siRNA selection using hybridization thermodynamics. *Nucleic Acids Research*, 36:640–647, 2008.
20. D. Lyumkis. Challenges and opportunities in cryo-EM single-particle analysis. *Journal of Biological Chemistry*, 294(13):5181–5197, 2019.
21. R. Madhugiri, M. Fricke, M. Marz, and J. Ziebuhr. Coronavirus cis-acting RNA elements. In *Advances in Virus Research*, volume 96, pages 127–163. Elsevier, 2016.
22. D. Mathews et al. Incorporating chemical modification constraints into a dynamic programming algorithm for prediction of RNA secondary structure. *PNAS*, 101(19): 7287–7292, 2004.
23. D. H. Mathews and D. H. Turner. Prediction of RNA secondary structure by free energy minimization. *Current Opinion in Structural Biology*, 16(3):270–278, 2006.
24. D. H. Mathews, J. Sabina, M. Zuker, and D. H. Turner. Expanded sequence dependence of thermodynamic parameters improves prediction of RNA secondary structure. *Journal of Molecular Biology*, 288(5):911–940, 1999.
25. D. H. Mathews, W. N. Moss, and D. H. Turner. Folding and finding RNA secondary structure. *Cold Spring Harbor Perspectives in Biology*, 2(12), 2010.
26. J. S. McCaskill. The equilibrium partition function and base pair probabilities for RNA secondary structure. *Biopolymers*, 29:1105–19, 1990.
27. J. Michalik. *Non-redundant sampling in RNA Bioinformatics*. PhD thesis, Université Paris-Saclay, 2019.
28. U. Mückstein, H. Tafer, J. Hackermüller, S. H. Bernhart, P. F. Stadler, and I. L. Hofacker. Thermodynamics of rna–rna binding. *Bioinformatics*, 22(10):1177–1182, 2006.
29. R. Nussinov and A. B. Jacobson. Fast algorithm for predicting the secondary structure of single-stranded RNA. *PNAS*, 77(11):6309–6313, 1980.
30. Y. Ponty. Efficient sampling of RNA secondary structures from the boltzmann ensemble of low-energy. *Journal of Mathematical Biology*, 56(1-2):107–127, 2008.
31. Y. Ponty and C. Saule. A combinatorial framework for designing (pseudoknotted) RNA algorithms. In *Int'l Workshop on Algorithms in Bioinformatics*, 2011.
32. R. Rangan, I. N. Zheludev, R. J. Hagey, E. A. Pham, H. K. Wayment-Steele, J. S. Glenn, and R. Das. RNA genome conservation and secondary structure in SARS-CoV-2 and SARS-related viruses: a first look. *RNA*, 26 (8):937–959, 2020.
33. RNAcentral Consortium et al. RNAcentral: a comprehensive database of non-coding RNA sequences. *Nucleic Acids Research*, 45(D1):D128–D134, 2017.
34. M. Sloma and D. Mathews. Exact calculation of loop formation probability identifies folding motifs in RNA secondary structures. *RNA*, 22(12), 2016.
35. H. Tafer, S. L. Ameres, G. Obernosterer, C. A. Gebeshuber, R. Schroeder, J. Martinez, and I. L. Hofacker. The impact of target site accessibility on the design of effective siRNAs. *Nature Biotech.*, 26(5):578–583, 2008.
36. F. Wu, S. Zhao, B. Yu, Y.-M. Chen, W. Wang, Z.-G. Song, Y. Hu, Z.-W. Tao, J.-H. Tian, Y.-Y. Pei, et al. A new coronavirus associated with human respiratory disease in China. *Nature*, 579(7798):265–269, 2020.
37. J. Zadeh, B. Wolfe, and N. Pierce. Nucleic acid sequence design via efficient ensemble defect optimization. *Journal of Computational Chemistry*, 32(3):439–452, 2010.
38. H. Zhang and S. Keane. Advances that facilitate the study of large RNA structure and dynamics by nuclear magnetic resonance spectroscopy. *Wiley Interdiscip. Reviews: RNA*, 10:e1541, 2019.
39. H. Zhang, L. Zhang, D. H. Mathews, and L. Huang. Linearpartition: linear-time approximation of RNA folding partition function and base-pairing probabilities. *Bioinformatics*, 36(Supplement_1):i258–i267, 2020.
40. J. Zhang and A. R. Ferré-D'Amaré. New molecular engineering approaches for crystallographic studies of large RNAs. *Curr. Opinion in Struct. Biol.*, 26:9–15, 2014.
41. J. Zuber, H. Sun, X. Zhang, I. McFadyen, and D. H. Mathews. A sensitivity analysis of RNA folding nearest neighbor parameters identifies a subset of free energy parameters with the greatest impact on RNA secondary structure prediction. *Nucleic Acids Research*, 45(10): 6168–6176, 2017.
42. M. Zuker and P. Stiegler. Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. *NAR*, 9(1):133–148, 1981.

Supporting Information

Improved and Linear-Time Stochastic Sampling of RNA Secondary Structure with Applications to SARS-CoV-2

He Zhang, Liang Zhang, Sizhen Li, David H. Mathews, and Liang Huang

```
function BEAMPRUNE(Z, j, b)
    candidates ← hash()                                  ▷ hash table: from candidates i to score
    for each i such that [i, j] in Z do
        candidates[i] ← Z(x_{1,i−1}) · Z(x_{i,j})        ▷ use Z(x_{1,i−1}) as prefix score
    candidates ← SELECTTOPB(candidates, b)               ▷ select top-b states by score
    for each i such that [i, j] in Z do
        if key i not in candidates then
            delete [i, j] from Z                         ▷ prune low-scoring states
```

Fig. SI 1. The BEAMPRUNE function from the Pseudocode of our main algorithm (Fig. 5).

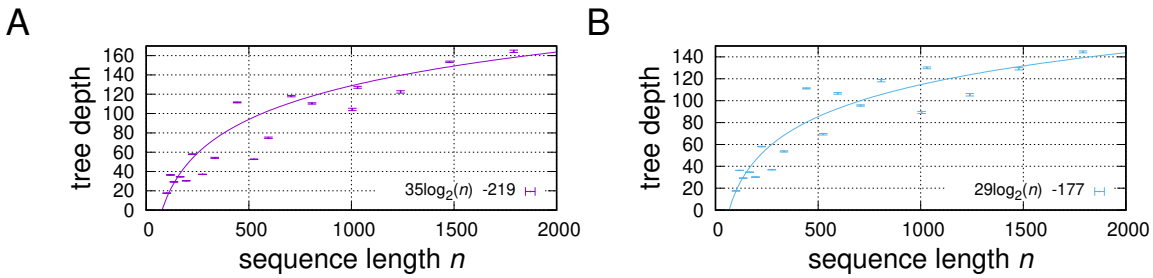

Fig. SI 2. The tree depth of a derivation against sequence length. The horizontal lines show variations across multiple samples (95% confidence intervals). **A**: the tree depth of the structures sampled by Vienna RNAsubopt. **B**: the tree depth of the structures sampled by LinearSampling.

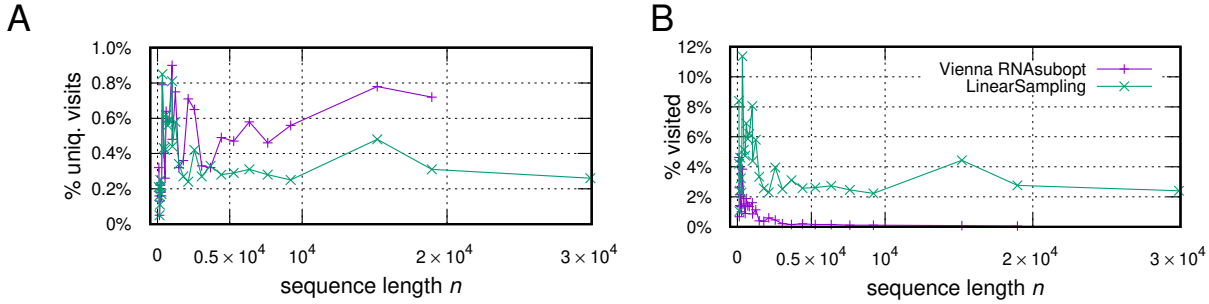

Fig. SI 3. Extension of Fig. 8 with varying sequence lengths. The sample size is 10,000. **A**: unique visit ratio against sequence length. **B**: visited ratio against sequence length.

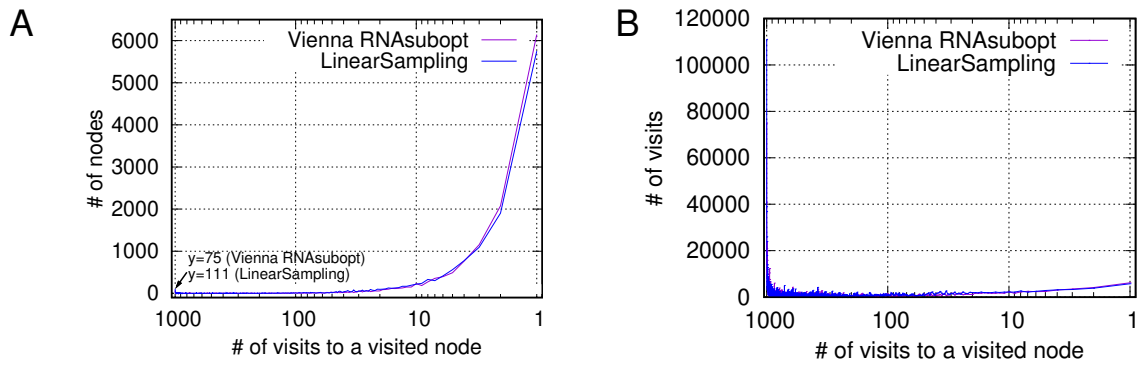

Fig. SI 4. Most of the visits are concentrated on a few nodes. **A**: the distribution of number of nodes against number of visits to a visited node, For example, 111 nodes are visited 1000 times in the sampling phase for LinearSampling. **B**: the distribution of number of visits against number of visits to a visited node.

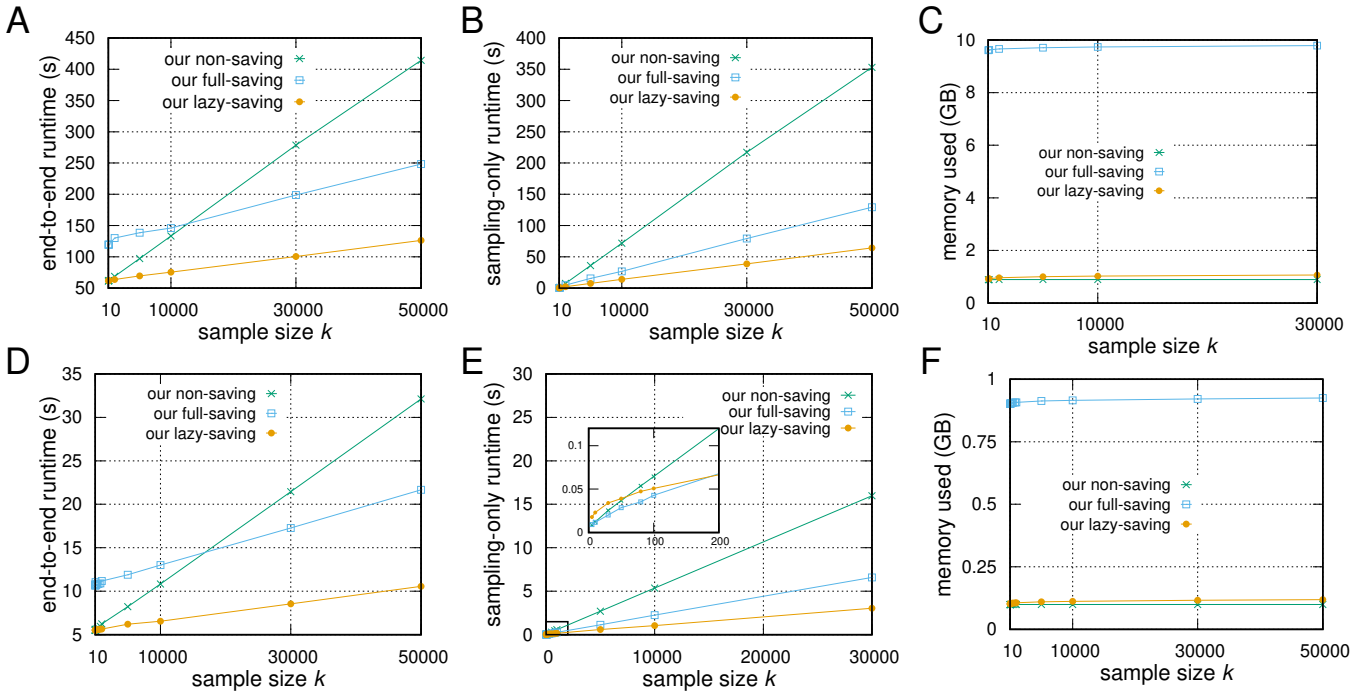

Fig. SI 5. Runtime and memory usage comparisons against sample size k, with different sequence lengths. **ABC**: SARS-CoV-2 reference sequence (29,903 *nt*). **DEF**: a 3,048 *nt* sequence in the RNAcentral dataset.

	E. coli	*T. thermophilus*	*S. aureus*	*B. subtilis*	*H. pylori*	overall	avg. runtime (s)
RNAsubopt	0.1317	0.1289	**0.1359**	0.1299	**0.1762**	0.1405	47.66
RNAsubopt (span=70)	0.1596	0.1530	0.1736	0.1630	0.2058	0.1710	77.36
RNAsubopt (span=150)	0.1404	0.1393	0.1554	0.1426	0.1864	0.1528	81.75
LinearSampling	**0.1185**	**0.1247**	0.1388	**0.1248**	0.1786	**0.1371**	8.08

Table SI 1. Accessibility defect and runtime comparisons between Vienna RNAsubopt, the RNAsubopt local sampling and LinearSampling on 23S rRNA family. The sample size is 10,000. Note that RNAsubopt local sampling is even slower.

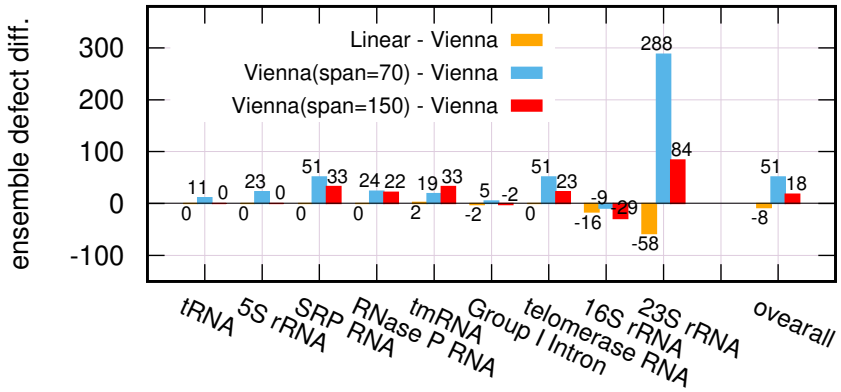

Fig. SI 6. The ensemble defect difference between LinearSampling and Vienna RNAsubopt, and between RNAsubopt local mode (maxBPspan=70 and maxBPspan=150) and RNAsubopt.

	tRNA	5S rRNA	SRP RNA	RNase P RNA	tmRNA	Group I Intron	telomerase RNA	16S rRNA	23S rRNA	overall
RNAsubopt	0.1826	0.1553	0.1309	0.2125	0.2245	0.2649	0.2998	0.1841	0.1405	0.1995
RNAsubopt (span=70)	0.2237	0.2097	0.1925	0.2380	0.2437	0.2623	0.3259	0.2006	0.171	0.2297
RNAsubopt (span=150)	0.1825	0.1553	0.1475	0.2270	0.2423	0.2607	0.3091	0.1828	0.1528	0.2067
LinearSampling	0.1823	0.1552	0.1309	0.2125	0.2249	0.2637	0.2994	0.1804	0.1371	0.1985

Table SI 2. Accessibility defect comparison between Vienna RNAsubopt, the RNAsubopt local sampling and LinearSampling on the ArchiveII dataset. The sample size is 10,000.

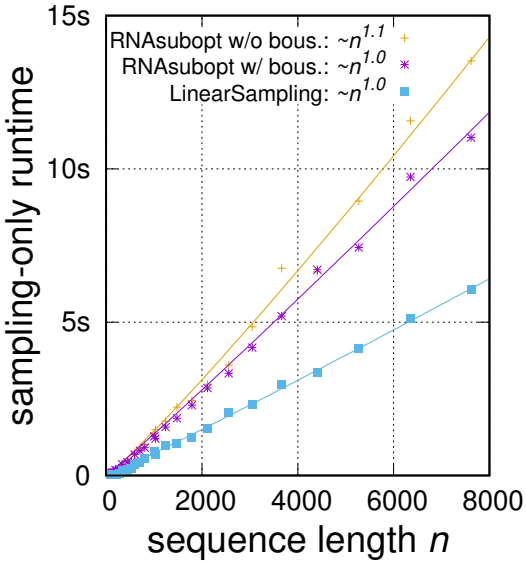

Fig. SI 7. The runtime of Vienna RNAsubopt (with or without the boustrophedon optimization (30)) and LinearSampling, on the RNAcentral dataset up to 8,000 *nt*.

start	subsequence	accessibility↓	gene
25703	CUUUUCUCUAUCUUU	0.992	ORF3a
25702	CCUUUUCUCUAUCUU	0.991	ORF3a
25704	UUUUCUCUAUCUUUA	0.991	ORF3a
25705	UUUCUCUAUCUUUAU	0.991	ORF3a
25701	CCCUUUUCUCUAUCU	0.976	ORF3a
25700	CCCCUUUUCUCUAUC	0.972	ORF3a
29074	AUACAAUGUAACACA	0.821	N
29075	UACAAUGUAACACAA	0.821	N
27132	UAUAAAUUAAACACA	0.791	M
27129	AACUAUAAAUUAAAC	0.780	M
27130	ACUAUAAAUUAAACA	0.780	M
27131	CUAUAAAUUAAACAC	0.780	M
23705	CCCACAAAUUUUACU	0.706	S
23706	CCACAAAUUUUACUA	0.706	S
23985	AUCCAUCAAAACCAA	0.623	S
9555	UUUACUCAUUCUUAC	0.604	ORF1ab
20149	AAUUAUUAUAAGAAA	0.580	ORF1ab
20148	CAAUUAUUAUAAGAA	0.576	ORF1ab
20147	UCAAUUAUUAUAAGA	0.557	ORF1ab
28433	ACCGCUCUCACUCAA	0.553	N
28691	AAUACACCAAAAGAU	0.553	N
28693	UACACCAAAAGAUCA	0.545	N
28692	AUACACCAAAAGAUC	0.540	N

Table SI 3. Regions of 15 nucleotides with negative log-odds of free energy change. They are found by LinearSampling running on the SARS-CoV-2 full genomes.