LinearSampling: 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, 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

ABSTRACT

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, e.g., for accessibility prediction. 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) much redundant work is repeatedly performed in the sampling phase; (c) the partition function runtime scales cubically with the sequence length. These issues prevent it from being used for full-length viral genomes such as SARS-CoV-2. To address these problems, we first present a hypergraph framework under which the sampling algorithm can be greatly simplified. We then present three sampling algorithms under this framework of which two eliminate redundant work in 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. For instance, LinearSampling is 111× faster (48s vs. 1.5h) than Vienna RNAsubopt on the longest sequence in the RNAcentral dataset that RNAsubopt can run (15,780 *nt*). More importantly, LinearSampling is the first sampling algorithm to scale to the full genome of SARS-CoV-2, taking only 96 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 (Eddy, 2001), catalyzing reactions (Doudna and Cech, 2002) and regulating diseases (Kung et al., 2013). Many functions of RNAs are highly related to their secondary structures. However, determining the structures using experimental methods, such as X-ray crystallography (Zhang and Ferré-D'Amaré, 2014), Nuclear Magnetic Resonance (NMR) (Zhang and Keane, 2019), or cryo-electron microscopy (Lyumkis, 2019), 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 (Nussinov and Jacobson, 1980, Zuker and Stiegler, 1981), but these methods do not capture the fact that multiple conformations exist at equilibrium, especially for mRNAs (Lai et al., 2018, Long et al., 2007, Lu and Mathews, 2008, Tafer et al., 2008). To address this, McCaskill (1990) pioneered the partition function-based methods, which account for the ensemble of all possible structures. The partition function can estimate the *base pairing probabilities* $p_{i,j}$, the probability of nucleotides i and j are paired, and the *unpaired probabilities* $q_i = 1 - \sum_j p_{i,j}$, the probability of i being 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. First, in many situations, we prefer to see a sample of representative structures according to their Boltzmann probabilities, which is more intuitive than the marginal probabilities. 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 the siRNA sequence design (Bohula et al., 2003, Kretschmer-Kazemi Far and Sczakiel, 2003, Petch et al., 2003). 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 (2003) pioneered the widely-used technique of stochastic sampling, which samples secondary structures according to

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

their probabilities in the ensemble. Their algorithm consists of two phases: the first "inside" phase computes the partition function (but not those marginal probabilities) in a standard bottom-up fashion, and the second "sampling" phase generates structures in a top-down iterative refinement fashion. This algorithm can solve the accessibility prediction problem easily: just sample k structures and see how many of them have the region of interest completely unpaired. Two popular RNA folding packages, RNAstructure (Mathews and Turner, 2006) and Vienna RNAfold (Lorenz et al., 2011), both implement this sampling 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. In their algorithm, each component, such as a base pair, hairpin loop, internal loop or multiloop, is sampled based on its conditional probability that represents its contribution to the partition function, and the corresponding probabilities are calculated using complicated equations. Secondly, in the sampling phase, much redundant work is repeatedly performed, wasting significant amount of time especially for large sample sizes. 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 whole-genome viral sequences such as SARS-CoV-2.

To alleviate these three issues, we present one solution to each of them. We adopt the hypergraph framework (Finkelstein and Roytberg, 1993, Gallo et al., 1993), under which the sampling algorithm can be greatly simplified. The key observation is that all needed information for sampling is encoded in the forward process of the equilibrium partition function computation. The information can be represented by a hypergraph, and then sampling can be simplified as recursive stochastic backtracing a path in the hypergraph.

Under this framework, we present three sampling algorithms, with the first (non-saving) being similar to but much simpler and cleaner than Ding and Lawrence's, and the other two (full-saving and lazy-saving) aiming to reduce the redundant work in the first.

Finally, we present LinearSampling, an end-to-end linear-time sampling algorithm that is orders of magnitude faster than the standard algorithm, to further overcome the slowness and scalability issues. We replace the classical cubic-runtime partition function computation with our recent proposed LinearPartition (Zhang et al., 2020), a linear-time algorithm that approximates the partition function, whose efficiency and approximation quality have been demonstrated. Following the hypergraph framework, we modify LinearPartition by saving the backpointers to build up the hypergraph in linear time against the sequence length, and sample structures simply by choosing a path in the hypergraph, which also takes linear time. With such efforts, LinearSampling achieves $111\times$ speedup compared to RNAsubopt on the longest sequence of length 15,780 *nt* that RNAsubopt can run.[1]

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 sample structures and calculate the accessibilities on such long sequences and consider global, long-range base pairs. We show that LinearSampling can successfully scale up to the whole-genome SARS-CoV-2 sequence of length about 30,000 *nt*, and list regions with high accessibilities, using the free energy change as an indicator.

2 Sampling Algorithms

We first formulate (in Sec. 2.2.1) the search space of RNA folding using the framework of (directed) hypergraphs (Finkelstein and Roytberg, 1993, Gallo et al., 1993) which have been used for both context-free parsing (Huang and Chiang, 2005, Klein and Manning, 2001), a problem closely related to RNA folding (Durbin et al., 1998), and RNA folding itself (Finkelstein and Roytberg, 1993, Ponty and Saule, 2011). This formulation makes it possible to present the various sampling algorithms succinctly, and simplifies complexity analysis.

In this framework, we will see (in Sec. 2.2) that sampling can be done in a simple way similar to the bottom-up inside phase for partition function, except that sampling is done in a top-down way; this formulation exploiting the symmetry is cleaner and simpler than the original algorithm of Ding and Lawrence (2003). In particular, we present three versions: non-saving (which is closest to Ding and Lawrence, i.e., it does not save hyperedges during the inside phase, and repeatedly recovers hyperedges during sampling phase), full-saving (which saves all hyperedges during inside phase), and lazy-saving (which only recovers and saves hyperedges on demand).

Finally, we present (in Sec. 2.3) our LinearSampling algorithm whose inside phase is very similar to LinearPar-

[1]The other popular tool RNAstructure takes 3+ days on this sequence because it includes coaxial stacking of helices as an additional folding feature.

tition in that both employ beam search to ensure linearity. The sampling phase of LinearSampling also benefits from beam search in the inside phase and also runs in linear time. Therefore, LinearSampling is the first algorithm to run in end-to-end linear-time without imposing constraints on base pair distance.

2.1 Hypergraph Framework

For an input RNA $\mathbf{x} = x_1 ... x_n$, 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 also define the **arity** of a hyperedge e, denoted $|e|$, to be the number of its children nodes ($|e| \triangleq |subs(e)|$). Hyperedges with 0, 1, or 2 arities are called *nullary, unary*, or *binary* hyperedges, resp. 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 (in dot-bracket format) from $subs$ into a structure for v. For example, a binary hyperedge might have $f(e)(a, b) = ab$. Each hyperedge e is also associated with an (extra) **energy term** $w(e) \in \mathbb{R}$. Here $\mathbb{S}$ and $\mathbb{R}$ are the sets of dot-bracket strings and reals, resp.

Now let us take the classical Nussinov algorithm (Nussinov and Jacobson, 1980) as a concrete example. For input sequence $\mathbf{x} = x_1 ... x_n$, 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 $\mathbf{x}_{i,j} = x_i ... x_j$ $(i \le j)$ which can be decomposed, and empty spans $\mathbf{x}_{i,i-1}$ $(i = 1...n)$ which are the terminal nodes. The Nussinov algorithm basically decomposes each non-empty span $\mathbf{x}_{i,j}$ in two ways: either base x_j is unpaired (unary) or paired with some x_k $(i \le k < j)$ (binary). Therefore, for each node, the incoming hyperedges are

$$\textsc{InEdges}(\mathbf{x}_{i,j}) = \begin{cases} \textsc{unary}(\mathbf{x}_{i,j}) \cup \textsc{binary}(\mathbf{x}_{i,j}) & i \le j \\ \{\langle [\,], \mathbf{x}_{i,i-1} \rangle\} & j = i-1 \end{cases}$$

where $\textsc{unary}(\mathbf{x}_{i,j}) = \{\langle \mathbf{x}_{i,j}, [\mathbf{x}_{i,j-1}] \rangle\}$ contains a single unary hyperedge, and its associated unary combination function $f_1(\cdot)$ is simply $f_1(a) = a$ **.** that appends an unpaired "**.**" for x_j to the substructure from $\mathbf{x}_{i,j-1}$:

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

Its associated energy term is $w = 0$ (kcal/mol) meaning no energy change. 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 $\mathbf{x}_{k+1,j-1}$:

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

All these hyperedges share the same combination function $f_2(\cdot, \cdot)$ which combines the two substructures along with the new (x_k, x_j) pair to form a structure for $\mathbf{x}_{i,j}$, i.e., $f_2(a, b) = a$ **(**b**)** . They also share $w = -1$ (kcal/mol), the stabilizing free energy term for forming a base pair.

At the end of the recursion, 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 just the whole sequence $\mathbf{x}_{1,n}$.

This framework can easily extend to other folding algorithms such as Zuker (Zuker and Stiegler, 1981) and LinearFold (Huang et al., 2019), 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 for $\mathbf{x}_{i,j}$, etc.

2.2 Non-Saving, Full-Saving, and Lazy Versions

Under the hypergraph framework, we first describe the standard "inside" phase for partition function calculation, and then present three variants of sampling, i.e., non-saving, full-saving, and lazy-saving in a unified way. While non-saving is more or less similar to the standard sampling algorithm, the other two are novel.

2.2.1. The Inside Phase. All sampling algorithms first calculate the partition function $Z(v)$ of each node v in the bottom up order (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

```
function INSIDE(V, saving)                    ▷ calculate partition function
  for each v in BOTTOMUPORDER(V) do ▷ traverse each node
    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 inside 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)
  INSIDE(V(x), False)                         ▷ do not save hyperedges
  for i = 1...k do SAMPLE(goal(V(x)))
```

Fig. 2. The non-saving version in the hypergraph framework. This version is similar to Ding and Lawrence (2003) but much simpler and cleaner due to the inside↔sampling symmetry.

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

Fig. 3. The full-saving version in the hypergraph framework.

```
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)
  INSIDE(V(x), False)                         ▷ do not save during inside
  visited ← ∅
  for i = 1...k do SAMPLELAZY(goal(V(x)), visited)
```

Fig. 4. The lazy version in the hypergraph framework.

phase. If we want to save all hyperedges (for the saving version) instead, we will need $O(n^3)$ space; the time complexity remains $O(n^3)$, but in practice the overhead for saving (line 8) is costly.

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 inside phase. When visiting a node v, it tries to sample a hyperedge e from v's incoming hyperedges $\text{INEDGES}(v)$ 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 *all* incoming hyperedges of v, though in the worst case it would (when $p \rightsquigarrow Z(v)$). It then backtraces recursively to the corresponding children node(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 (i.e., the recursion tree) 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).[2] Our experiments in Sec. 3 (Fig. SI 6) show that, again 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.[3]

This version is closest to the original Ding and Lawrence (2003) algorithm, but simpler and cleaner. Our key idea is to exploit the structural symmetry between the inside 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). This formulation results in unnecessarily complicated implementations (see Vienna RNAsubopt for an example).[4] Ponty (2008) analyzes the special case of sampling under the Nussinov model by exploiting the symmetry (though his work could have been generalized

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

[3] Ponty (2008) applies the "Boustrophedon" method to reduce the worst-case time also to $O(n \log n)$. Our experiments (Fig. SI 6) show that it does further improve the runtime, but only slightly.

[4] We point out that RNAstructure (Mathews et al., 1999)'s sampling is similar to our non-saving version except for being non-recursive.

to other systems), but we are the first to formulate general sampling (Nussinov, Zuker, LinearFold, etc.) under a unified framework that exploits symmetry, and unlike his work, our experiments are on the full Turner energy model rather than the simplified Nussinov-Jacobson one.

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, each hyperedge might be recovered multiple times across different samples, esp. when the sample size k is large. This begs the question: why don't we save all hyperedges during the inside phase, so that no hyperedge needs to 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 $O(kn)$.[5]

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. A key observation is that nodes are not backtraced with equal chance. Actually, the vast majority of nodes are *never* visited during the sampling phase even for large sample size (our experiments in Fig. 7 show that only $< 0.5\%$ nodes are visited for 20,000 samples of a 3,048 *nt* length sequence with Vienna RNAsubopt), so most of the saving is 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.

[5]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" (Cormen et al., 2009).

```
1:  function LinearInside(x, b, saving)          ▷ b is the 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. The pseudocode of a simplified version of the LinearSampling algorithm on the Nussinov-Jacobson energy model. 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.

The complexity analysis of lazy-saving is more involved. It depends on how many unique nodes are visited (and need to have hyperedges recovered and saved) among k samples. Given such correlation, we denote

$$g(k) \triangleq \frac{\text{\# of uniq. nodes visited}}{\text{\# of nodes visited}} \cdot k$$

to be the "effective sample size" (or "effective k"). In other words, $g(k)$ characterizes the actual amount of hyperedge-recovering work for generating k samples (in non-saving, $g(k) = k$ but in lazy-saving, $g(k) \ll k$). Thus, for the lazy-saving, $O(g(k)n)$ unique nodes are visited (as each sample is of size $O(n)$) and $O(g(k)n^2)$ hyperedges are recovered in the worst-case, resulting in a total worst-case runtime of $O(g(k)n^2 + kn \log n)$, where the second term is the runtime of sampling saved hyperedges. Similarly, the best-case runtime, where the derivation tree is balanced, is $O(g(k)n \log n + kn)$, fol-

		inside time	inside 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		$O(n^2)$	$O(g(k)n^2 + kn \log n)$	$O(g(k)n \log n + kn)$	$O(g(k)n^2)$
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		$O(nb)$	$O(h(k)nb + kn \log b)$	$O(h(k)n \log b + kn)$	$O(h(k)nb)$

Table 1. Time and space complexities for exact and linear-time sampling with non-, full- and lazy-saving versions (n is the sequence length, k is sample size, $g(k)$ and $h(k)$ are "effective sample sizes", and b is beam size; see text for definitions).

lowing full-saving's analysis. Regarding the sampling-phase space complexity, no extra space is required for non-saving or full-saving, while $O(g(k)n^2)$ is needed for lazy-saving in the worst-case (see Table 1 for a summary).

2.3 The LinearSampling Algorithm

The biggest bottleneck of the standard sampling algorithm is actually not redundant calculations, but the cubic runtime of the inside (partition-function) phase, which prevents it from scaling to full-length viral genomes such as SARS-CoV-2. To address this problem, we present LinearSampling, an end-to-end linear-time sampling algorithm, by replacing the $O(n^3)$ inside phase with our linear-time approximate algorithm, LinearPartition (Zhang et al., 2020), followed by one of the three versions (non-, full-, and lazy-saving) for the sampling phase.

Fig. 5 describes a simplified LinearSampling 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 (Zhang et al., 2020). The inside 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 by the following analysis (see Tab 1). The binary search time to sample a saved hyperedge changes from $O(\log n)$ to $O(\log b)$ since at most b hyperedges are saved for each node. Therefore, the worst-case and best-case sampling runtimes become (details omitted due to space limit) $O(h(k)nb + kn \log b)$ and $O(h(k)n \log b + kn)$, resp., where $h(k)$ is the "effective sample size" in LinearSampling (corresponding to $g(k)$ in exact sampling). Note that $g(k)$ and $h(k)$ are both in the range of $[1, \min(k, n)]$. In Fig. 7 we plot $g(k)$ and $h(k)$ for sequences with different length, and we confirm that they are both much smaller than either k or n.

3 Results

3.1 Efficiency and Scalability

We benchmark the runtime and memory usage on a collected dataset, which consists of 34 sequences sampled from RNAcentral (RNAcentral Consortium et al., 2017). We evenly split the range from 0 to 35,000 into 60 bins by log-scale. For each bin we randomly select at most one sequence (some bins are empty); within 100 *nt* only one sequence is chosen. We refer this dataset as the RNAcentral dataset in the paper. For benchmarks, We use a Linux machine (CentOS 7.7.1908) with 2.30 GHz Intel Xeon E5-2695 v3 CPU and 755 GB memory, and gcc 4.8.5.

3.1.1. Comparing the Non-Saving, Full-Saving and Lazy-Saving Versions. First, we compare the three versions of the LinearSampling algorithm: non-saving, full-saving and lazy-saving.

Fig. 6A-C show the comparisons of runtime and memory usage against sequence length between the three versions. For end-to-end comparison (Fig. 6A), lazy-saving is the fastest, followed by full-saving, and then by non-saving. This confirms our analysis: (1) for a relatively large sample size (e.g., 10,000), saving the hyperedges leads to a speedup by avoiding unnecessary recovering; and (2) the lazy mechanism only saves the hyperedges on demand, and gets rid of the cost of saving "less weighted" hyperedges, which leads to another speedup. For sampling-only runtime (Fig. 6B), it is clear that the non-saving version is the slowest, while the full-saving and the lazy-saving versions are almost identical. Fig. 6C confirms that the full-saving requires the most memory, while non-saving reduces about 90% of the memory usage; the lazy-saving version is between the other two, but closer to the non-saving curve.

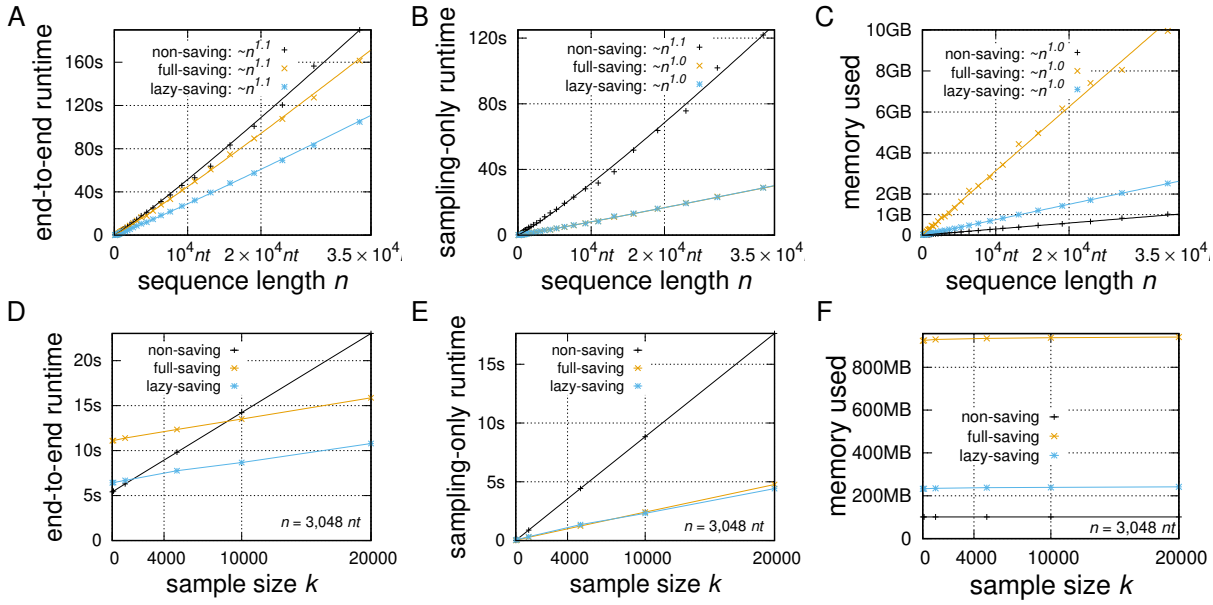

Fig. 6. Runtime and memory usage comparisons against sequence length n (**A**-**C**) and against sample size k (**D**-**F**), between non-saving, full-saving, lazy-saving versions on the RNAcentral dataset. The sample size is 10,000. **A** and **D**: end-to-end runtime. **B** and **E**: sampling-only runtime. **C** and **F**: memory usage. Note that **D**-**F** are based on a 3,048 *nt* sequence from the RNAcentral dataset. Fig. SI 4 shows sequences with different lengths, which have similar results.

Fig. 6D-F compare the runtime and memory usage against sample size with a 3,048 *nt* sequence. Regarding the end-to-end comparison, non-saving is the fastest for small sample size (e.g., 1,000), but is the slowest when sample size exceeds around 9,000. The curve of non-saving intersects with lazy-saving at around 1,500, and intersects with full-saving at around 9,000; between the two points of intersection, the lazy-saving version is the fastest. For the sampling-only comparison, lazy-saving is close to full-saving, and slightly faster with sample size larger than 8,000. If considering the memory usage, non-saving uses the least; lazy-saving doubles the usage; full-saving increases the number by about 9 times.

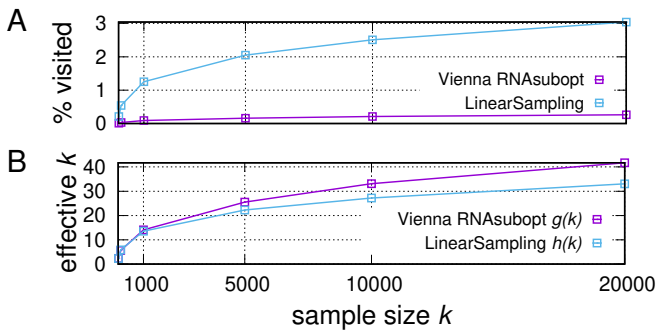

Fig. 7. In practice only a small portion of nodes are visited many times. The curves are based on a sequence of 3,048 *nt*. Please refer to Fig. SI 3 for the trend with sequence length. **A**: the percentage of visited nodes against k. **B**: the number of unique visited nodes divided by the number of visited nodes, i.e., "effective k" ($g(k)$ in purple and $h(k)$ in blue).

In the sampling phase, though extra work occurs for recovering hyperedges, lazy-saving is in practice as fast as full-saving because only a small portion of the nodes are visited, which is confirmed in Fig. 7A. We can see that the percentage of visited nodes is always smaller than 0.3% and 3% for RNAsubopt and LinearSampling, respectively, when $k \leq 20,000$, and grows slower and slower against sample size. LinearSampling's percentage is larger due to the pruning of nodes with small $Z(v)$. Fig. 7A clearly shows that it is a waste to save the majority of hyperedges that are barely recovered, which can be addressed by lazy-saving. On the other hand, lazy-saving is better than non-saving since it avoids a large number of re-calculations during sampling phase. Fig. 7B shows $g(k)$, number of unique visited nodes divided by number of visited nodes for RNAsubopt ($h(k)$ for LinearSampling), is much smaller than both k and n.

From Fig. 6–7, it is clear that the lazy-saving version is the fastest among 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.

3.1.2. Comparing Vienna RNAsubopt and LinearSampling. Next, we compare the efficiency and scalability between LinearSampling and the baseline, Vienna RNAsubopt (Lorenz et al., 2011). We benchmark them on the RNAcentral dataset with sample size 10,000.

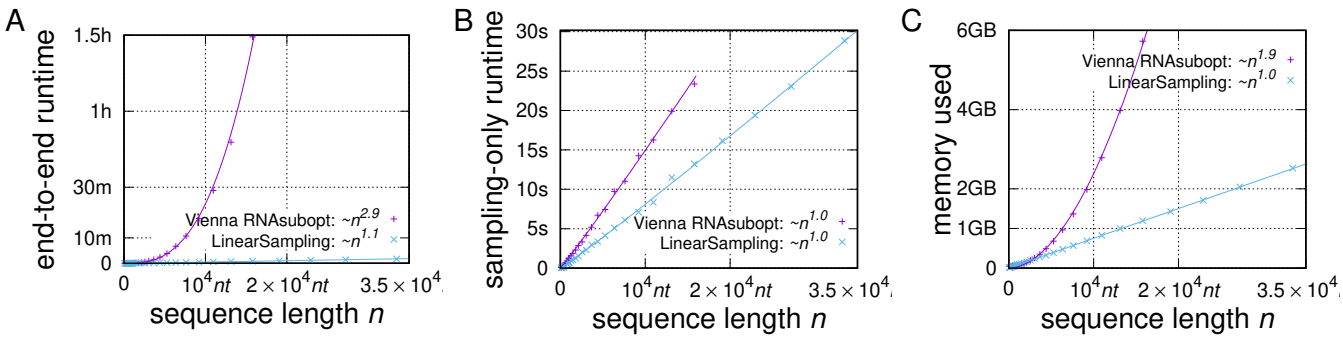

Fig. 8. Runtime and memory usage comparisons between RNAsubopt and LinearSampling on the RNAcentral dataset. The sample size is 10,000. **A**: end-to-end runtime, including the inside and the sampling phases. **B**: sampling-only runtime. **C**: memory usage of RNAsubopt and LinearSampling. Note that LinearSampling can scale to longer sequences.

Fig. 8A and B show the comparisons in two dimensions: the end-to-end runtime and the sampling-only runtime. In the former case (Fig. 8A), LinearSampling scales almost linearly against sequence length, and is faster than RNAsubopt; in the latter case (Fig. 8B), LinearSampling takes strictly linear runtime and is nearly twice as fast. We find that RNAsubopt overflows on a sequence of length 19,071 *nt*, while LinearSampling scales up to longer sequences, e.g., the longest sequence (33,461 *nt*) in the dataset. This suggests that LinearSampling is able to be applied to the full-length of SARS-CoV-2 genomes (about 30,000 *nt*). Compared to the longest sequence (15,780 *nt*) RNAsubopt can run in the RNAcentral dataset, LinearSampling is 111× faster (48 seconds vs. 89 minutes).

Fig. 8C confirms that the memory usage of LinearSampling is also linear, but RNAsubopt requires nearly quadratic memory size against sequence length. LinearSampling only uses 1.2 GB memory for the sequence of length 15,780 *nt*, while RNAsubopt uses 5.7 GB.

3.2 Quality of the Samples

In this subsection, we investigate the quality of the samples, using the ArchiveII dataset (Mathews et al., 1999, Sloma and Mathews, 2016), which contains a set of sequences with well-determined structures. We follow the preprocessing steps of a previous study (Zhang et al., 2020), which leads to a subset of 2,859 sequences with reliable structures distributed in 9 families.

3.2.1. Approximation Quality to Base Pairing Probabilities. To evaluate the sampling quality, i.e., how well it approximates to the ensemble distribution, Ding and Lawrence (2003) investigated the frequency of the MFE structures 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 structures is extremely small for long sequences, e.g., 23S rRNA.

Alternatively, we investigate the root-mean-square deviation (RMSD) between the probability matrices $p(S)$, which is derived from the sample set S, and p', which is generated by Vienna RNAfold or LinearPartition.

The RMSD is defined as:

$$\mathrm{RMSD}(p, p') = \sqrt{\frac{1}{|\mathrm{pairings}(\mathbf{x})|} \sum_{(i,j)\in\mathrm{pairings}(\mathbf{x})} (p_{i,j}(S) - p'_{i,j})^2}$$

where $\mathrm{pairings}(\mathbf{x})$ is the set of all possible Watson-Crick and G-U pairs on a sequence $\mathbf{x}$, and $p_{i,j}(S)$, the probabilities of i pairing with j, is calculated from S as:

$$p_{i,j}(S) = \frac{1}{|S|} \sum_{\mathbf{y}\in S} \mathbb{1}[(i,j) \in \mathrm{pairs}(\mathbf{y})]$$

where $\mathrm{pairs}(\mathbf{y})$ is the set of base pairs in structure $\mathbf{y}$.

Fig. 9A shows three curves of average RMSD against sample size on the ArchiveII dataset. The green curve illustrates the RMSD of base pairing probabilities generated by LinearSampling and Vienna RNAfold. Even though LinearSampling approximates the partition function based on LinearPartition, which introduces a small change in the base pairing probabilities (Zhang et al., 2020), the RMSD is only 0.015 with sample size 10, and drops down to around 0.005 quickly with sample size 5,000. We further investigate the RMSD between LinearSampling and LinearPartition (the blue curve), and RNAsubopt and RNAfold (the purple curve). We observe that the two curves are almost identical, suggesting LinearSampling can generate structures strictly matching with the ensemble distribution as RNAsubopt. It is clear that the RMSD shrinks when sample size increases. They both start at around 0.012 with a sample size 10, then go down and are far below 0.004 with 100 sampled structures, and further approximate to 0 slowly with a 20,000

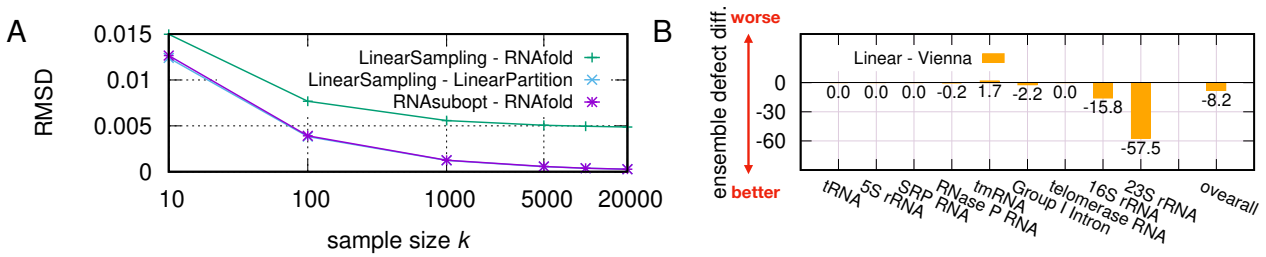

Fig. 9. LinearSampling matches with the ensemble distribution, and is better correlated to the ground truth structure. **A**: the root-mean-square deviation (RMSD) against sample size. The RMSD is between the base pairing probabilities directly generated by RNAfold (or LinearPartition) and by RNAsubopt (or LinearSampling) sampled structures. The RMSD is averaged on 9 families in the ArchiveII dataset. **B**: the ensemble defect difference of each family. The overall difference is averaged by families.

sample size. We notice that $k = 10,000$ is a reasonable sample size since the RMSD (about 0.0004) is already close enough to 0.

3.2.2. Correlation with the Ground Truth Structure. Next, we investigate the sampled structure's correlation with the ground truth structure. We use ensemble defect (Zadeh et al., 2010), the expected number of incorrectly predicted nucleotides over the ensemble, which can be formalized as:

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

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. 9B shows the ensemble defect difference between RNAsubopt and LinearSampling on each family and overall. Note that better correlation to the ground truth structures requires lower ensemble defect. The families are ordered in their average sequence length, from the shortest to the longest. 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 short families. But on 16S and 23S rRNAs, LinearSampling has lower ensemble defect, confirming it performs better on longer sequences. In Fig. SI 5 we also present the comparison of RNAsubopt local mode, with a base pair length limitation of 70. Note that RNAsubopt local mode does not have a default window size, and we choose 70 following the default setting in RNAplfold (Bernhart et al., 2006). It is obvious that the local sampling has much higher (worse) ensemble defect on 23S rRNA, since it ignores all base pairs longer than 70 *nt*.

	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.

An important application of sampling algorithm is to calculate region's accessibility. Therefore, we also evaluate the accessibilities of window size 4 (Ding and Lawrence, 2003) derived from the two systems based on the ground truth structures. We denote it as the *accessibility defect*, $D(S, \mathbf{y}^*)$, and define it as:

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

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

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

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

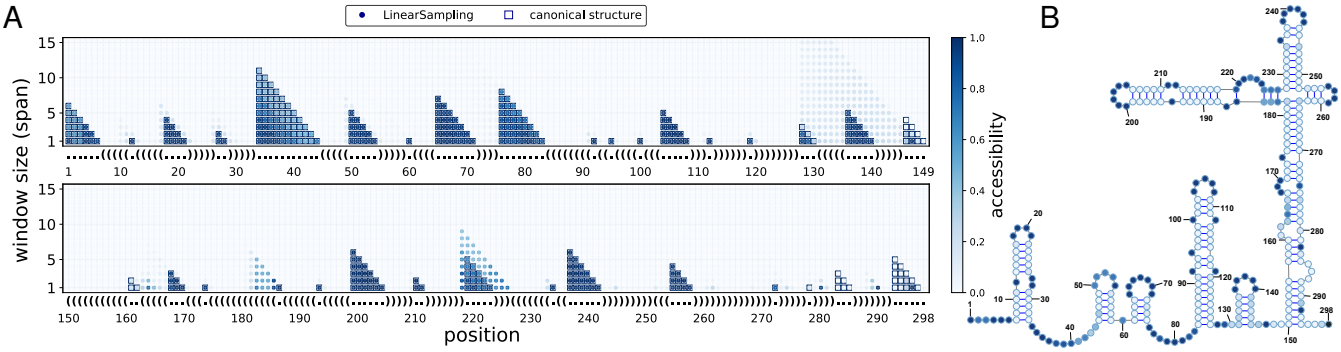

Fig. 10. The accessibilities derived from LinearSampling correlate well with the unpaired region in the canonical structure of SARS-CoV-2 5'-UTR (Madhugiri et al., 2016). Note that the full sequence was used for the accessibility calculation, but we only illustrate the 5'-UTR region in the figure because 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**.

able to scale, for the first time, up to the whole-genome of SARS-CoV-2, and predict its accessible regions.

We run LinearSampling on NC_0405512.2 (Wu et al., 2020), the reference sequence of SARS-CoV-2. 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 (Madhugiri et al., 2016). We sample 10,000 structures, calculate the accessibilities and compare them with the canonical structure. Fig. 10A 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., the bottom row in Fig. 10A) on the 5'-UTR structure.

Secondly, we aim to obtain potentially accessible regions. A previous study (Rangan et al., 2020) 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 (Do et al., 2006) 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. We use the free energy change ΔG, which is defined formally as:

$$\Delta G = -RT \log \frac{k_0}{k - k_0}$$

where k is the sample size, k_0 is the number of accessible samples for a given window size w (i.e., the nucleotides within the w are all unpaired), R is the universal gas constant and T is the thermodynamic temperature. We set the window size to be 15 following Rangan et al. (2020). We only show the fragments whose free energy changes are negative, i.e., the corresponding accessibilities are larger than 0.5, meaning they are more likely to be opening than closing. We list all 23 regions found by LinearSampling in Table 3. Some of the regions are overlapped, resulting in a total of 9 separate accessible regions. We illustrate them in Fig. 11. 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 (Huang et al., 2020).

4 Discussion

In this paper, we focus on simplifying and accelerating the stochastic sampling algorithm for a given RNA sequence. Algorithmically, we present a hypergraph frame-

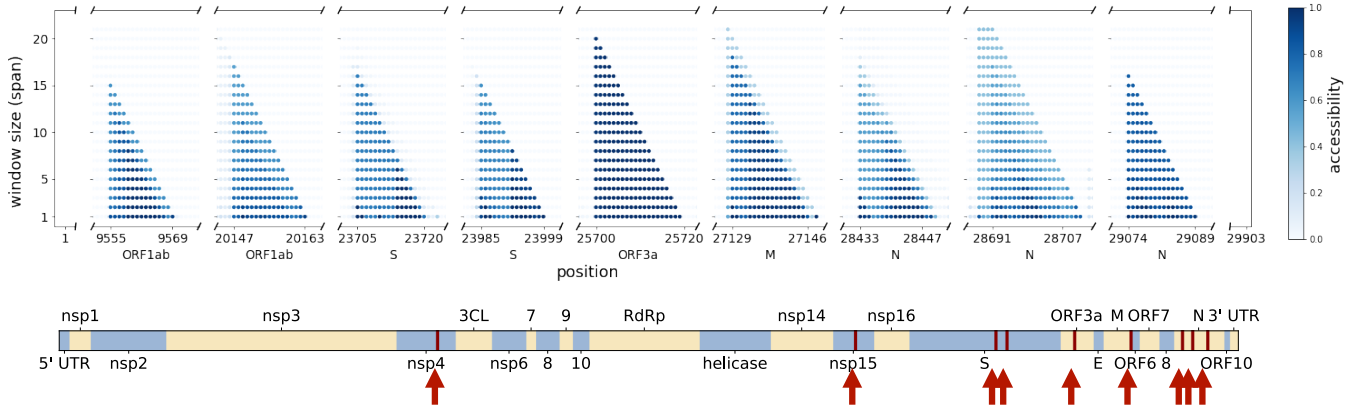

Fig. 11. LinearSampling predicts 9 accessible regions in SARS-CoV-2 full genomes. Note that ORF1ab can be further divided into smaller genes, e.g., nsp4 and RdRp.

start	subsequence	log-odds	accessibility↓	gene
25703	CUUUUCUCUAUCUUU	-2.96	0.992	ORF3a
25702	CCUUUUCUCUAUCUU	-2.93	0.991	ORF3a
25704	UUUUCUCUAUCUUUA	-2.91	0.991	ORF3a
25705	UUUCUCUAUCUUUAU	-2.91	0.991	ORF3a
25701	CCCUUUUCUCUAUCU	-2.29	0.976	ORF3a
25700	CCCCUUUUCUCUAUC	-2.18	0.972	ORF3a
29074	AUACAAUGUAACACA	-0.94	0.821	N
29075	UACAAUGUAACACAA	-0.94	0.821	N
27132	UAUAAAUUAAACACA	-0.82	0.791	M
27129	AACUAUAAAUUAAAC	-0.78	0.780	M
27130	ACUAUAAAUUAAACA	-0.78	0.780	M
27131	CUAUAAAUUAAACAC	-0.78	0.780	M
23705	CCCACAAAUUUUACU	-0.54	0.706	S
23706	CCACAAAUUUUACUA	-0.54	0.706	S
23985	AUCCAUCAAAACCAA	-0.31	0.623	S
9555	UUUACUCAUUCUUAC	-0.26	0.604	ORF1ab
20149	AAUUAUUAUAAGAAA	-0.20	0.580	ORF1ab
20148	CAAUUAUUAUAAGAA	-0.19	0.576	ORF1ab
20147	UCAAUUAUUAUAAGA	-0.14	0.557	ORF1ab
28433	ACCGCUCUCACUCAA	-0.13	0.553	N
28691	AAUACACCAAAAGAU	-0.13	0.553	N
28693	UACACCAAAAGAUCA	-0.11	0.545	N
28692	AUACACCAAAAGAUC	-0.10	0.540	N

Table 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.

work 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 during the inside phase and avoids re-computing for sampling, and the lazy-saving that only recovers and saves hyperedges on demand. Then we propose LinearSampling, an end-to-end linear-time stochastic sampling algorithm, which employs beam search to ensure linearity.

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.

ACKNOWLEDGMENTS. This work is supported in part by National Institutes of Health (R01 GM076485 to D.H.M.)

References

S. H. Bernhart, I. L. Hofacker, and P. F. Stadler. Local RNA base pairing probabilities in large sequences. *Bioinformatics*, 22(5):614–615, 2006.

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.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, Cambridge, US, 2009.

Y. Ding and C. E. Lawrence. A statistical sampling algorithm for RNA secondary. *Nucleic Acids Research*, 31(24):7280–7301, 2003.

C. Do, D. Woods, and S. Batzoglou. CONTRAfold: RNA secondary structure prediction without physics-based models. *Bioinformatics*, 22(14):e90–e98, 2006.

J. A. Doudna and T. R. Cech. The chemical repertoire of natural ribozymes. *Nature*, 418(6894):222–228, 2002.

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

S. R. Eddy. Non-coding RNA genes and the modern RNA world. *Nature Reviews Genetics*, 2(12):919–929, 2001.

A. Finkelstein and M. Roytberg. Computation of biopolymers: a general approach to different problems. *BioSystems*, 30 (1-3):1–19, 1993.

G. Gallo, G. Longo, and S. Pallottino. Directed hypergraphs and applications. *Discrete Applied Mathematics*, 42(2):177–201, 1993.

L. Huang and D. Chiang. Better k-best parsing. *Proceedings of the Ninth International Workshop on Parsing Technologies*, pages 53–64, 2005.

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, 07 2019. ISSN 1367-4803.

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.

D. Klein and C. D. Manning. Parsing and Hypergraphs. In *Proceedings of the Seventh International Workshop on Parsing Technologies (IWPT-2001), 17-19 October 2001, Beijing, China*, 2001.

R. Kretschmer-Kazemi Far and G. Sczakiel. The activity of siRNA in mammalian cells is related to structural target accessibility: a comparison with antisense oligonucleotides. *Nucleic Acids Research*, 31(15):4417–4424, 2003.

J. T. Y. Kung, D. Colognori, and J. T. Lee. Long noncoding RNAs: Past, present, and future. *Genetics*, 193(3):651–669, 2013.

W.-J. Lai et al. mRNAs and lncRNAs intrinsically form secondary structures with short end-to-end distances. *Nature Communications*, 9(1):4328, 2018.

D. Long et al. Potent effect of target structure on microRNA function. *Nature Structural & Molecular Biology*, 14(4): 287–294, 2007.

R. Lorenz et al. ViennaRNA package 2.0. *Algorithms for Molecular Biology*, 6(1):1, 2011.

Z. J. Lu and D. H. Mathews. Efficient siRNA selection using hybridization thermodynamics. *Nucleic Acids Research*, 36: 640–647, 2008.

D. Lyumkis. Challenges and opportunities in cryo-EM single-particle analysis. *Journal of Biological Chemistry*, 294(13): 5181–5197, 2019.

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.

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.

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.

J. S. McCaskill. The equilibrium partition function and base pair probabilities for RNA secondary structure. *Biopolymers*, 29:1105–19, 1990.

R. Nussinov and A. B. Jacobson. Fast algorithm for predicting the secondary structure of single-stranded RNA. *Proceedings of the National Academy of Sciences, U.S.A.*, 77(11): 6309–6313, 1980.

A. K. Petch, M. Sohail, M. D. Hughes, I. Benter, J. Darling, E. M. Southern, and S. Akhtar. Messenger RNA expression profiling of genes involved in epidermal growth factor receptor signalling in human cancer cells treated with scanning array-designed antisense oligonucleotides. *Biochemical Pharmacology*, 66(5):819–830, 2003.

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.

Y. Ponty and C. Saule. A combinatorial framework for designing (pseudoknotted) rna algorithms. In *International Workshop on Algorithms in Bioinformatics*, pages 250–269. Springer, 2011.

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.

RNAcentral Consortium et al. RNAcentral: a comprehensive database of non-coding RNA sequences. *Nucleic Acids Research*, 45(D1):D128–D134, 2017.

M. Sloma and D. Mathews. Exact calculation of loop formation probability identifies folding motifs in RNA secondary structures. *RNA*, 22(12), 2016.

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 Biotechnology*, 26(5):578–583, 2008.

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.

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.

H. Zhang and S. Keane. Advances that facilitate the study of large RNA structure and dynamics by nuclear magnetic resonance spectroscopy. *Wiley Interdisciplinary Reviews: RNA*, 10:e1541, 04 2019.

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.

J. Zhang and A. R. Ferré-D'Amaré. New molecular engineering approaches for crystallographic studies of large RNAs. *Current Opinion in Structural Biology*, 26:9—15, June 2014. ISSN 0959-440X.

M. Zuker and P. Stiegler. Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. *Nucleic Acids Research*, 9(1):133–148, 1981.

Supporting Information

LinearSampling: 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).

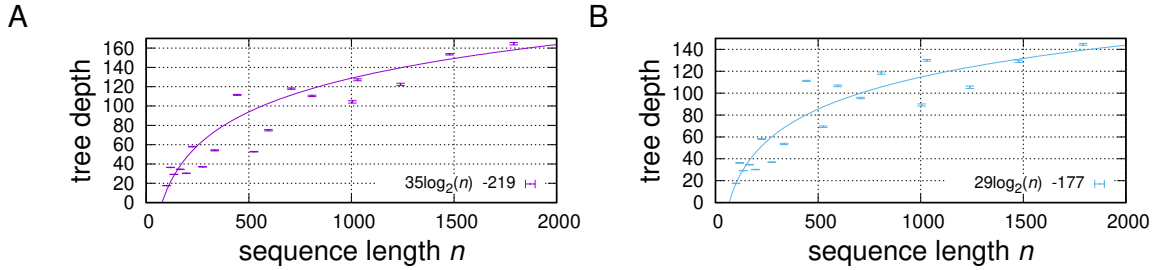

Fig. SI 2. The tree depth of a derivation against sequence length. **A**: the tree depth of the structures sampled by Vienna RNAsubopt. **B**: the tree depth of the structures sampled by LinearSampling.

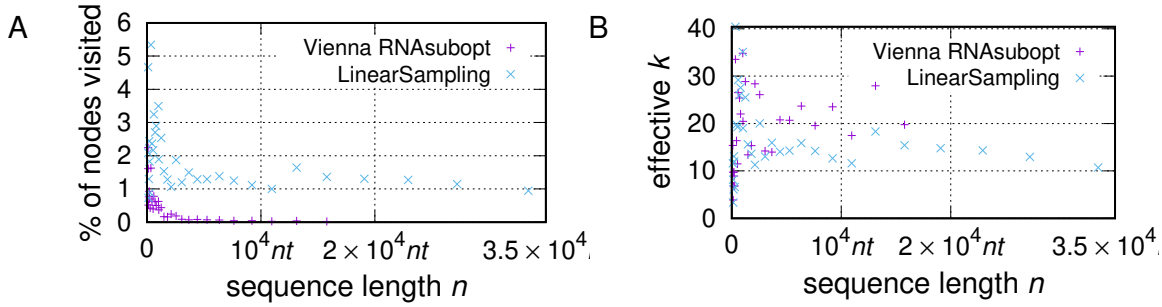

Fig. SI 3. Extension of Fig. 7 with varying sequence lengths. The sample size is 1,000. **A**: the percentage of visited nodes against sequence length. **B**: $g(k)$ and $h(k)$ against sequence length. Note that RNAsubopt overflows on a sequence of length 19,071 *nt*, so we do not plot the data point above 19,071 *nt* for RNAsubopt.

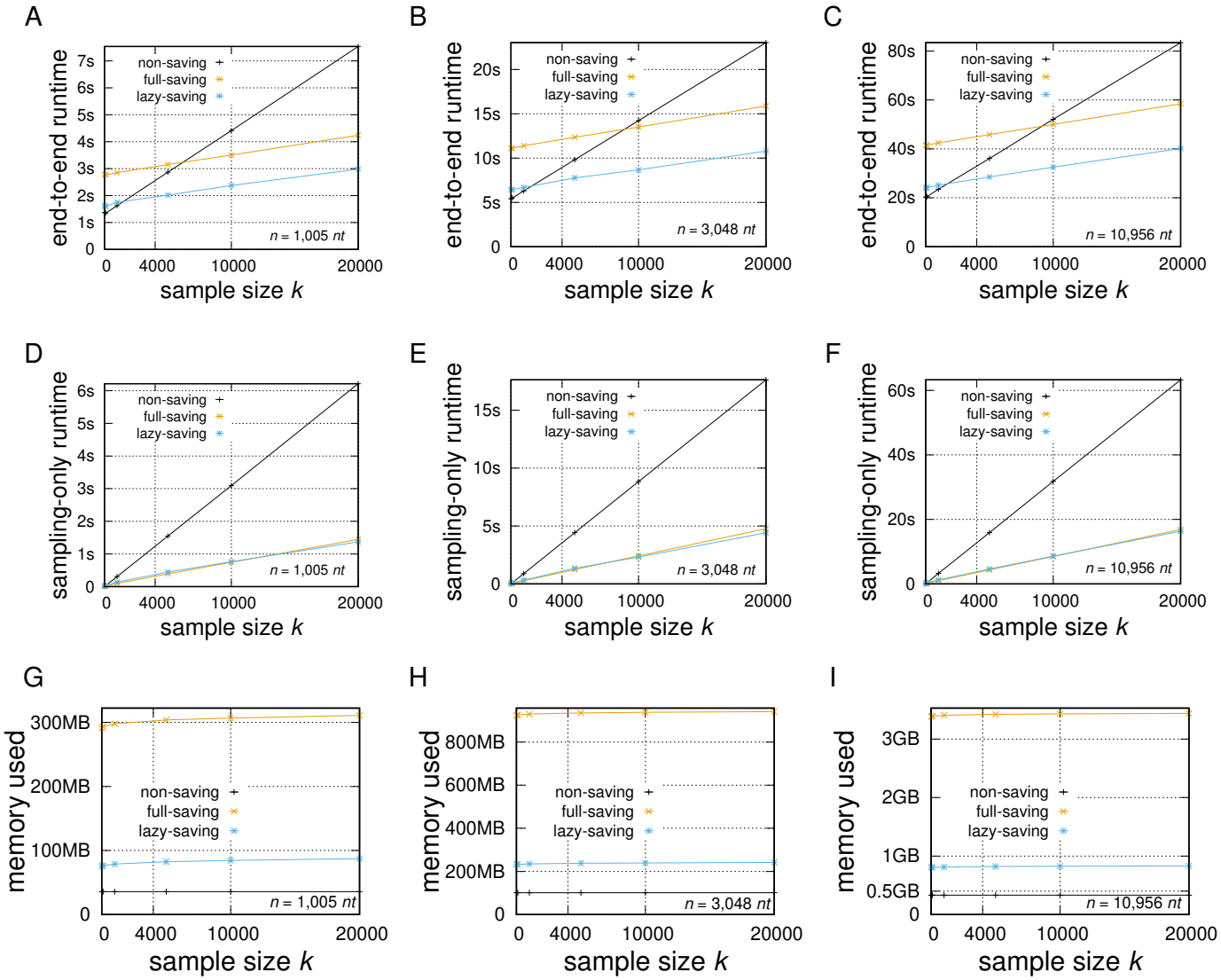

Fig. SI 4. Runtime comparisons against sample size k, with different sequence lengths (**ADG**: 1,005 *nt*; **BEH**: 3,048 *nt*; **CFI**: 10,956 *nt*). **A–C**: end-to-end runtime. **D–F**: sampling-only runtime. **G–I**: memory usage.

	E. coli	*T. thermophilus*	*S. aureus*	*B. subtilis*	*H. pylori*	overall
RNAsubopt	0.1317	0.1289	**0.1359**	0.1299	**0.1762**	0.1405
RNAsubopt (local; window size 70)	0.1596	0.1530	0.1736	0.1630	0.2058	0.1710
RNAsubopt (local; window size 150)	0.1404	0.1393	0.1554	0.1426	0.1864	0.1528
LinearSampling	**0.1185**	**0.1247**	0.1388	**0.1248**	0.1786	**0.1371**

Table SI 1. Accessibility defect comparison between Vienna RNAsubopt, the local sampling and LinearSampling on 23S rRNA family. The sample size is 10,000.

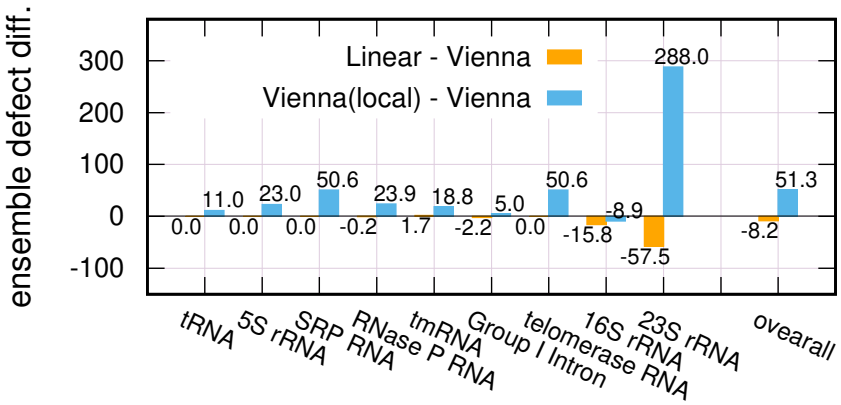

Fig. SI 5. The ensemble defect difference between LinearSampling and Vienna RNAsubopt, and between RNAsubopt local mode (maxBPspan=70) and RNAsubopt.

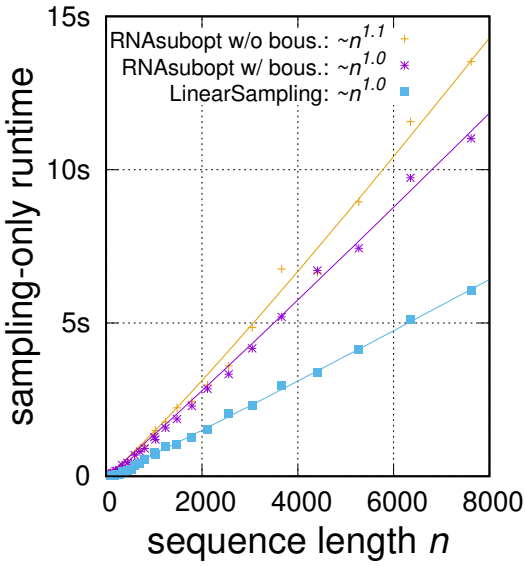

Fig. SI 6. The runtime of Vienna RNAsubopt (with or without the boustrophedon optimization (Ponty, 2008)) and LinearSampling, on the RNAcentral dataset up to 8,000 *nt*.