Developing complex RNA design applications in the Infrared framework

Hua-Ting Yao, Yann Ponty, Sebastian Will

Developing complex RNA design applications in the Infrared framework

Hua-Ting Yao[1,2], Yann Ponty[1], and Sebastian Will[1]

[1] LIX, CNRS UMR 7161, Ecole Polytechnique, Institut Polytechnique de Paris, Palaiseau, France
[2] School of Computer Science, McGill University, Montreal, Canada

Abstract. Applications in biotechnology and bio-medical research call for effective strategies to design novel RNAs with very specific properties. Such advanced design tasks require support by computational design tools but at the same time put high demands on their flexibility and expressivity to model the applications-specific requirements. To address such demands, we present the computational framework Infrared. It supports developing advanced customized design tools, which generate RNA sequences with specific properties, often in a few lines of Python code. This text guides the reader in tutorial-format through the development of complex design applications. Thanks to the declarative, compositional approach of Infrared, we can describe this development as step-by-step extension of an elementary design task. Thus, we start with generating sequences that are compatible with a single RNA structure and go all the way to RNA design targeting complex positive and negative design objectives with respect to single or even multiple target structures. Finally, we present a 'real-world' application of computational RNA design of a biotechnological device. We use Infrared to generate design candidates of an artificial AND-riboswitch, which could activate gene expression (only) in the simultaneous presence of two different metabolites.

1 Introduction

Designing molecules with novel functionality or very specific desirable properties for applications in biological fundamental research, biotechnology, and medicine, is a highly complex task that typically requires interdisciplinary efforts, combining biochemical experimentation and computational design. Compared to proteins, designing RNAs can be particularly attractive for the construction of new biotechnological devices. On the one hand, functional RNA molecules save the detour of translation into proteins, and can therefore act more efficiently, e.g. as fast on/off-switches of gene activity. On the other hand, the design process itself can build on the well-understood combinatorics of RNA secondary structure and available computational models and algorithms.

Still, the supporting RNA design computationally is highly demanding. First of all, RNA design is an optimization problem with often complex objectives with respect to multiple (secondary) structures, e.g. when the designed RNAs should

switch between alternative structural states or fold via specific intermediary structures. Moreover, RNA design is computationally complex even in simple problem variants. For example, one cannot efficiently design an RNA that preferentially folds into a single given target structure in the nearest-neighbor energy model, since this problem is NP-hard.

Here we present the framework Infrared, which addresses the multiple demands of computational RNA design in several ways:

- To address the issues of computational complexity, it follows the classical decomposition of RNA design into two related sub-problems, often called **positive and negative design**. Positive RNA design aims at very specific properties (e.g. specific energy) for certain 'target' RNA structures, while negative design additionally aims to avoid similar properties for all (exponentially many) other, 'non-target' structures. While efficient algorithms for the latter problem can not exist, the system automatically derives fixed-parameter tractable algorithms to solve the (sub-problem) positive design efficiently. In many cases, this can already solve negative design by searching through relatively few samples of good positive designs. To address harder negative design tasks, we demonstrate constraint generation as well as classic stochastic optimization techniques as supported by the system.
- Real-world RNA design applications typically demand for targeting thermodynamic criteria referring to multiple target structures (e.g. on and off-states of riboswitches, binding pockets of aptamers and competing structures in specific energetic relations), potentially under side constraints like moderate GC-content or avoidance of specific sequence motifs. Thus, we design Infrared as a library that supports programmers to develop complex and potentially novel design strategies based on a declarative constraint framework. This allows application programmers to harness the power of fixed-parameter tractable sampling of designs in an easy to use system.
  In this chapter, we guide the reader through the use of this library and show by examples how to develop and run design programs from IPython notebooks. Further examples, covering as well non-design applications, are provided as part of the documentation of Infrared. In addition it should be mentioned that we published complete design tools based on the library for the specific applications of the generation of multi-target designs (RNARedPrint v2, as reimplementation of RNARedPrint [7]) and negative RNA design (RNAPOND [13]), which we hope are both directly useful to some readers. These tools can as well serve as further examples for the use of Infrared in command line tools.

Let us illustrate some fundamental ideas of the Infrared framework with a first simple example. After importing Infrared

```
1 from infrared import *
2 from infrared.rna import *
```

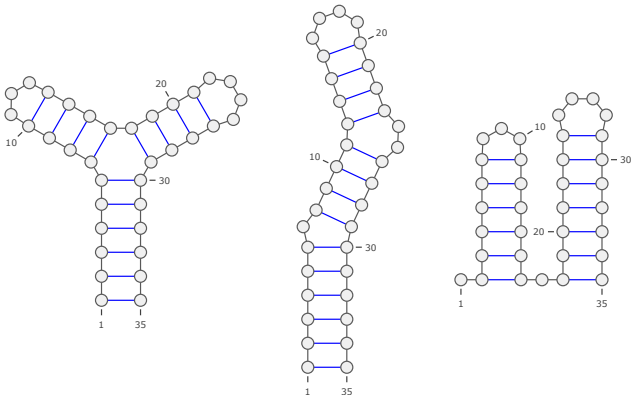

Fig. 1. Three RNA target structures for RNA design, which will serve us as running examples. They are shown in 2D representation (by `VARNA` [3]) and as dot-bracket strings. The latter represent base pairs by balanced parentheses.

a few lines of code let us generate 10 random (uniformly sampled) designs that can fold into the first structure of Figure 1.

```
target = "((((((((((...))))(((((....)))))))))))"
model = Model(len(target), 4)
model.add_constraints(BPComp(i, j) for (i, j) in parse(target))
sampler = Sampler(model)
samples = [sampler.sample() for _ in range(10)]
```

While this code doesn't yet come close to exploiting the power of the system, it demonstrates the main structure of using Infrared's FPT sampling engine. After a short prelude and defining the input, the users set's up a constraint model, here defining that we want to sample one of 4 nucleotides for each sequence position. Moreover by the constraints `BPComp`, we require the nucleotides of base pairs to be complementary. Finally, we generate a sampler for this constraint model, which specifies the entire problem, and generate 10 samples.

A slight extension of this example provides a first glimpse at the possibilities in Infrared. Generating uniformly sampled sequences for multi-target design, analogously to RNABluePrint [6], requires only a slight extension of the above model. To target the three structures of Figure 1, we define

```
targets = ["((((((((((...))))((((....))))))))))",
           "((((((.(((((((((....))))..))))))))))",
           ".((((((...))))))..(((((((....)))))))"]
```

Then, we simply add constraints for all these structures to our model. For this purpose, we rewrite line 6 of the previous example to loop over the targets

```
for target in targets:
    model.add_constraints(BPComp(i, j)
                          for (i, j) in parse(target))
```

and generate samples from this model in exactly the same way as before. The involved computation and algorithmic structure required by the multi-target case is handled transparently "under the hood". Internally, the problem is solved by an efficient algorithm that automatically adapts in complexity to the dependency network due to the multiple target structures.

Chapter overview. For preparation, we describe the installation of the library and recommended additional software. In Section Methods, we systematically develop a complex multi-target design application using the Infrared framework. We start by targeting positive design objectives in increasingly complex settings. This is followed by a discuss of several approaches that integrate negative design objectives. Thus, we show-case constraint generation as we applied in RNAPOND, stochastic optimization based on the library and finally, a full-fledged application to the design of an AND-riboswitch.

2 Material: Installing Infrared

We recommend installing Infrared using the package manager Conda as described below. Alternatively but less conveniently, the software can be compiled and installed from its source found on Gitlab `https://gitlab.inria.fr/amibio/Infrared`. In this chapter, we describe the release 1.0 of the software.

Package manager Conda installation. Unless Conda is already installed on your system, we recommend to install it in the form of Miniconda from `https://conda.io/en/latest/miniconda.html`. The page contains installation instructions for Windows, MacOs and Linux.

Infrared installation. To use Conda, it typically has to be activated by running the shell command

```
conda activate
```

in a terminal. All required and recommended software can be installed from the command line by

```
conda install -c conda-forge -c bioconda 'infrared=1.0b' viennarna
↪  jupyter
```

The command installs the packages `infrared`, `viennarna`, and `jupyter`; the flags `-c conda-forge -c bioconda` specify the required channels. For the largest part of the tutorial, only the package **infrared** is required. For energy evaluation and RNA structure prediction, which we use for advanced, negative design, we utilize the Vienna RNA package [10]. As of yet, it can not be installed under Windows via Conda; Windows users must therefore remove **viennarna** from the installation command. Note that there is as well no other convenient way to install the *Python interface* to the Vienna RNA package on Windows. Nevertheless, most of the the examples can be run without Python bindings to the Vienna RNA package. For some of the advanced design examples, we provide a work around, which will allow Windows users to run the examples, after installing the package using its Windows installer (`https://www.tbi.univie.ac.at/RNA/#download`, select operating system "Windows" to obtain the download link).

3 Methods

One can run all code of this tutorial from an IPython notebook using using **jupyter** (or a comparable system). The tutorial notebook is available from Infrared's Gitlab repository `https://gitlab.inria.fr/amibio/Infrared/-/blob/master/Doc/Bookchapter%20Tutorial.ipynb`. Using `jupyter-notebook`, it is loaded by the command

```
jupyter-notebook "Bookchapter Tutorial.ipynb"
```

3.1 Elementary use of Infrared—A simple design model

Recall the first example from the introduction. We want to generate sequences compatible with a single target structure. In Infrared this idea can be expressed in the form of a constraint/function network model.

For this purpose, given a target length n

```
n = 35
```

we set up a new model with `n` variables $X_0, \ldots, X_{n-1}$ that each can take one of four different values (representing the nucleotides A,C,G,U by integers 0,1,2,3).

```
model = Model(n, 4)
```

This is simply the Infrared-way of modeling the solutions of the design task, namely nucleotide sequences of length n. Next, we restrict the solution space by enforcing compatibility to a target structure (again from the running example). This is accomplished by adding a complementarity constraint for each base pair

```
target = "((((((((((...))))((((....))))))))))"
model.add_constraints(BPComp(i, j) for (i, j) in parse(target))
```

The constraint `BPComp(i, j)` is *valid* (or *satisfied*) if the variables X_i and X_j represent complementary nucleotides (i.e. one pair of A-U, C-G, G-C, G-U, U-A, U-G). Technically, solutions are assignments that assign to each of the variables $X_0, \ldots, X_{n-1}$ a value of its domain $\{0, 1, 2, 3\}$; moreover they must be *valid* assignments that satisfy all the constraints.

Once we have a model, which defines the solution space, we can instantiate a sampler

```
sampler = Sampler(model)
```

which allows us to draw sample solutions by calls to `sample()`, e.g. we obtain 10 sampled designs by

```
samples = [sampler.sample() for _ in range(10)]
```

Recall that nucleotides are represented by numbers, such that we typically want to convert sampled assignments to sequences (over A, C, G, U); for this purpose, one typically applies Infrared's function `ass_to_seq()`, like

```
sequences = [ass_to_seq(sample) for sample in samples]
```

Without further specifications, such solutions will be sampled from the uniform distribution.

3.2 Imposing additional constraints—Sequence constraints in IUPAC code

As one of the main of the constraint paradigm, more complex tasks can be specified compositionally. This allows us to tailor the solution space very naturally by imposing additional constraints. In design problems it is often useful to express prior knowledge on the sequence design space in IUPAC nucleotide encoding. To provide just one (arbitrary) example, we could specify the loop regions as RYY and GNRA, as well as the left and right most base as strong (S) by the constraint string

```
iupac_sequence = "SNNNNNNNNNRYYNNNNNNNNGNRANNNNNNNNNS"
```

Corresponding constraints can then be added to our model by

```
for i, x in enumerate(iupac_sequence):
    model.add_constraints(ValueIn(i, iupacvalues(x)))
```

Here, we make use of Infrared's function `iupacvalues()` which returns the possible nucelotides values allowed by a IUPAC symbol and the constraint `ValueIn` which constrains the allowed values of a variable.

3.3 Functions and features—Control of GC content

Infrared offers many ways to build on this initial design model and tailor it towards more sophisticated design and very specific applications. For a start, we want to generate sequences with control over their GC content.

In Infrared, we can define the GC content as a feature of solutions, which itself is specified by a group of functions. For this purpose, we extend the previous model by

```
model.add_functions([GCCont(i) for i in range(n)], 'gc')
```

This defines one function `GCCont` for each variable (or nucleotide) and add them to the model (all in the same function group named `gc`). Each single function has value 1, if the respective variable X_i represents C or G and value 0, otherwise. Infrared automatically derives a feature with name `gc` that sums over all these function values. Thus, it counts the G and C nucleotides (reflecting the GC content as the *feature value*).

Shifting the GC content distribution. Infrared offers two major ways to make use of this feature. In the simpler case, one can control the sample distribution by setting the weight of the feature before constructing the sampler and drawing samples, e.g.

```
model.set_feature_weight(1, 'gc')
sampler = Sampler(model)
samples = [sampler.sample() for _ in range(1000)]
```

Plotting sequence logos and histograms of the GC contents of these samples shows that it is no longer uniformly distributed, but the sample distribution is shifted towards higher GC content. Negative weights shift to lower GC content, while 0 keeps the distribution uniform (Fig. 2). Technically, the samples s are generated from a Boltzmann distribution, depending on the weight w_{gc} of feature gc, i.e. their probabilities are proportional to the Boltzmann weights based on their GC contents $\mathrm{GC}(s)$:

$$exp(w_{\mathrm{gc}} \cdot \mathrm{GC}(s)).$$

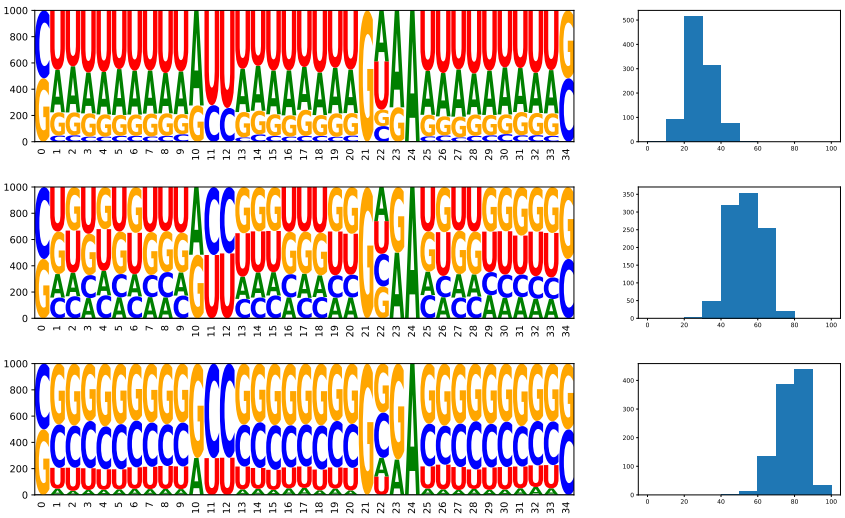

Fig. 2. Sequence logos and GC-content histograms for each time 1000 samples with respective weights -1, 0, and 1 (top, middle, bottom) of the feature gc. The sequences are compatible to the first example target structure and the IUPAC string SNNNNNNNNNRYYNNNNNNNNGNRANNNNNNNNNS.

Excursion 1: a first glimpse at the Infrared sampling engine. Having seen a few sampling examples, we are ready to talk about the computations behind the scenes. First observe that variables not involved in base pairs can apparently be sampled independently (and in a rather straightforward manner: either uniformly or with probability proportional to their Boltzmann factor).

It's more interesting how to sample variables X_i and X_j connected by a base pair (i, j). Since determining the variables simultaneously does not generalize well, we handle one after the other. Choosing the value of the variable X_i before X_j requires looking ahead to the possible solutions in combination with X_j. In uniform sampling, we can choose based on the number of possible solutions for each value in X_i's domain. For example, choosing G leaves two values for X_j to satisfy the complementarity constraint, whereas C leaves exactly one; consequently, G must be selected two times as often as C if we want to sample uniformly. Remarkably, sampling from a Boltzmann distribution works in the same way but 'counting' solutions weighted by their Boltzmann factors. Conversely, uniform sampling turns out to be as a special case of Boltzmann sampling with weight zero.

To inform the choice of X_i (before choosing X_j), we marginalize the sums of Boltzmann weights $\exp(w_{\mathrm{gc}} \cdot \mathrm{GC}(s))$ over the possible values of X_j. In other words, we utilize partial partition functions.

Value of X_i	Possible values of X_j	Partition function (GC content)
A	U	$\exp(w_{\mathrm{gc}} \cdot 0)$
C	G	$\exp(w_{\mathrm{gc}} \cdot 2)$
G	C, U	$\exp(w_{\mathrm{gc}} \cdot 2) + \exp(w_{\mathrm{gc}} \cdot 1)$
U	A, G	$\exp(w_{\mathrm{gc}} \cdot 0) + \exp(w_{\mathrm{gc}} \cdot 1)$

For sampling, the value of X_i is selected with probability proportional to its corresponding partition function from the table. After X_i is determined, the choice of a value for X_j becomes comparably simple.

For the previous examples, Infrared's solving mechanism indeed boils down to fixing an arbitrary order of the variables and precomputing partition functions as described for the variables in base pairs. In the presence of more complex dependencies between variables, Infrared still chooses values for the variables one-by-one in a predetermined optimized variable order. Given this order, it precomputes partial partition functions to derive the probabilities to choose values in order to sample from the desired Boltzmann distribution. The key to efficient uniform sampling in Infrared is thus to precompute such partition functions as efficiently as possible. After the precomputation the choice for each variable is performed in constant time, resulting in linear time sampling. We are going to discuss further details of the Infrared engine, when we progress to network models with more complex dependencies as they result from simultaneously targeting several RNA structures.

Targeting specific GC content. Let us return to the control of the GC content. Instead of direct tweaking of the weight, we can even ask Infrared to target a

specific feature value. For example, we generate targeted samples with a GC content of $75\% \pm 1\%$ from our model by

```
sampler = Sampler(model)
sampler.set_target(0.75*n, 0.01*n, 'gc')
samples = [sampler.targeted_sample() for _ in range(1000)]
```

Note the use of `Sampler`'s method `targeted_sample()` in place of `sample()`. This method provides access to an automatic mechanism that returns only samples within the tolerance from the target. To make such a rejection strategy effective, we iteratively sample, estimate the current mean of the feature value and then update the feature weight. Concretely, Infrared implements a form of multi-dimensional Boltzmann sampling [2] as applied in RNARedPrint [7].

3.4 Controlling energy—Multiple features

While, for instructional purposes, we first presented how to target GC content, an even more obvious target of RNA design is the energy of the target structure—or in other words, the affinity of the designs to the target structure.

Similar to the GC content, RNA energy can be modeled as a sum of function values. This holds even for the detailed nearest-neighbor energy model of RNAs, where energy is composed of empirically determined or trained loop energies [12,1]. Here, we focus on the much simpler base pair energy model, which has been demonstrated to be an effective proxy for the Turner (nearest neighbor) model in design applications [7].

In this simple model, every type of base pair (A-U, C-G or G-U) receives a different energy. To define the feature `energy`, we impose a function for each base pair `(i, j)` in the target structure and moreover distinguish terminal and non-terminal base pairs, simply by `(i-1, j+1) not in bps`. Infrared provides a default parameterization, which has been originally trained for use with RNARedPrint [7].

```
bps = parse(target)
model.add_functions([BPEnergy(i, j, (i-1, j+1) not in bps)
                     for (i, j) in bps], 'energy')
```

In the same way as for the feature `gc`, we can set the weight of the new feature `energy` and in this way shift the distribution of base pair energies. In Infrared, can even simultaneously control the energy and the GC content, since it samples (depending on the weights w_{gc} and w_{energy}) from the two-dimensional Boltzmann distribution, where probabilities are proportional to

$$\exp(w_{\text{gc}} \cdot \text{GC}(s) + w_{\text{energy}} \cdot \text{BPEnergy}(s)).$$

We observe that, for a fixed target structure, the energies of sequences in the base pair energy model are strongly correlated to their energies in the Turner model. Thus shifting the distribution of base pair energies indirectly shifts the distribution of nearest neighbor energies.

For example, we generate sequences with high affinity to the target structure and specific GC content, straightforwardly extending previous code and ideas:

```
model.set_feature_weight(-2 'energy')
sampler = Sampler(model)
sampler.set_target(0.75*n, 0.01*n, 'gc')
samples = [sampler.targeted_sample() for _ in range(10)]
```

We arrived at the functionality of IncARNation [11] as application of the Infrared library, requiring only a few lines of Python code. We remark that IncARNation implements the stacking energy model, which is slightly more complex than the base pair model. This model is as well available in Infrared, but was not shown to have clear advantages for sampling. To use it, we would simply replace adding the `BPEnergy` functions by

```
model.add_functions([StackEnergy(i, j)
    for (i, j) in bps if (i+1, j-1) in bps], 'energy')
```

3.5 Targeting Turner energies—Customized features

This correlation between the base pair and Turner energy models can be moreover exploited in Infrared to directly target Turner energies. We will make use of the Vienna RNA package to evaluate the Turner energies using its function energy_of_struct.

```
from RNA import energy_of_struct
```

Note that importing this module fails on Windows users, but we provide a workaround in the tutorial notebook.

Now, we simply add a custom feature for Turner energy that controls the group of functions `energy`:

```
model.add_feature('Energy', 'energy',
    lambda sample, target=target:
        energy_of_struct(ass_to_seq(sample), target))
```

Consequently, we can target specific (realistic) Turner energy of the target structure (and simultaneously target specific GC content)

```
sampler = Sampler(model)
sampler.set_target(0.75*n, 0.01*n, 'gc')
sampler.set_target(-10, 0.5, 'Energy')
samples = [sampler.targeted_sample() for _ in range(10)]
```

This example provides a first demonstration of the targeting flexibility due to Infrared multi-dimensional Boltzmann sampling engine, which can simultaneously target several features. We already see how this generalizes over advanced design tools like IncARNation.

3.6 Multiple-target structures—Complex dependencies

Due to the compositionality of constraints (and functions) in Infrared, defining multiple targets does not look any different than defining a single target structure. Thus, let us right away define model to target the structures of Fig. 1, which were previously defined in the list `targets` with length `n`.

```
model = Model(n,4)
for k, target in enumerate(targets):
    bps = parse(target)
    model.add_constraints(BPComp(i, j) for (i, j) in bps)
    model.add_functions([BPEnergy(i, j, (i-1, j+1) not in bps)
                         for (i, j) in bps], f'energy{k}')
model.add_functions([GCCont(i) for i in range(n)], 'gc')
```

Note how we just add constraints and functions for each target structure, but define different names for their functions groups (`energy0`, `energy1`, `energy2`), such that we could control them separately. As well, note that we are going to control GC content. Here, we could add further constraints like the ones for a specific IUPAC sequence.

By now, it will appear natural to the attentive reader that we can go on by defining Turner energy features and specific targets.

```
for k, target in enumerate(targets):
    model.add_feature(f'Energy{k}', f'energy{k}',
        lambda sample, target=target:
            energy_of_struct(ass_to_seq(sample), target))
sampler = Sampler(model)
```

```
sampler.set_target(0.75*n, 0.01*n, 'gc')
sampler.set_target( -15, 1, 'Energy0')
sampler.set_target( -20, 1, 'Energy1')
sampler.set_target( -20, 1, 'Energy2')
```

Finally, as expected, Infrared will indeed generate sequences that are compatible to all structures and hit the prescribed target energies and GC content.

```
samples = [sampler.targeted_sample() for _ in range(10)]
```

At this point, we arrived at reimplementing the essential functionality of RNARedPrint [7] in the Infrared framework. A full-fledged Infrared-based implementation with command line interface is moreover provided, as RNARedPrint 2.x, at `https://gitlab.inria.fr/amibio/RNARedPrint`.

Excursion 2: a deeper dive into Infrared's sampling engine Setting several target structures and targeting specific energies for them seems natural due to Infrared's modeling syntax, nevertheless it is not at all obvious *a priori* how the system effectively generates solutions that satisfy the constraints and target specific properties.

Generation of samples from a multi-dimensional Boltzmann distribution. For generating samples, Infrared implements a general solving strategy based on tree decompositions and cluster tree elimination (CTE) [8]. Such techniques have been well known in the (larger) context of constraint processing [4]; more recently, we described this approach specialized to multi-target RNA design for our approach RNARedPrint. The cluster tree elimination scheme yields a fixed-parameter tractable algorithm to compute (partial) partition functions, which let's us generate samples from a multi-dimensional Boltzmann distribution. In our example, this means that we can efficiently generate samples with probabilities proportional to

$$\exp(w_{\mathrm{gc}} \cdot \mathrm{GC}(s) + \sum_{k=0}^{2} w_{\mathrm{energy}k} \cdot \mathrm{energy}k(s)).$$

Due to the CTE scheme, the computation is based on a tree decomposition of the network of the dependencies induced by the constraints and functions in the model (aka *dependency graph*); see Figure 3. The concept of tree decomposition allows us to recursively compute partial partition functions for the subtrees of the tree decomposition by processing the variables in its bags in bottom-up order.

Moreover, this computation can be performed efficiently due to dynamic programming (which tabulates partial result that would otherwise be re-computed redundantly).

After all partition functions are computed, each sample is generated in a backtrace running from the root to the leaves. In this way, whenever a new variable is introduced in the depth-first/top-down traversal of the tree decomposition, its value can be chosen with the correct probability to generate the desired multi-dimensional Boltzmann distribution.

This solving strategy explains why Infrared can sample from one network faster than from the other. Tree-like networks of dependencies are processed quickly, while complex cyclic dependencies require tree decompositions with more variables per bag, since valid tree decompositions must satisfy certain conditions w.r.t. the dependencies in the network (which in turn guarantee the correctness of the dynamic programming evaluation).

Finally, since the computation requires to enumerate all possible sub-assignments in each bag, the computation time is exponential in the maximum number of variables per bag—this complexity is commonly described in terms of tree width, which is defined as this number minus 1.

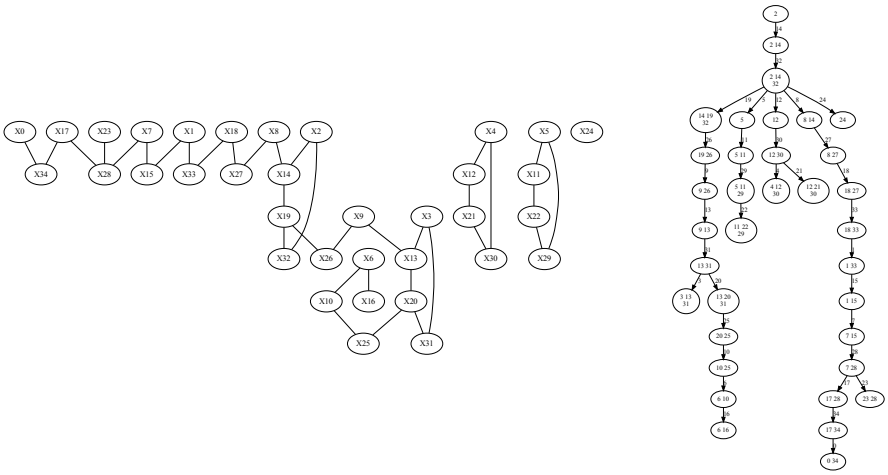

Fig. 3. (Left) Dependency graph of the multi-target design model showing the dependencies between the variables $X0, \ldots, X34$ of this model **(Right)** A tree decomposition of this graph that puts the variables into bags (only variable indices shown). The directed edges are labeled with the indices of variables introduced by the child bag. This decomposition has a tree width of two, since it's largest bags contain three (tree width plus one) variables.

Targeting specific properties. For targeting very specific properties like certain GC content and energies of the target structures, Infrared utilizes the just described sampling engine to iteratively sample from a multi-dimensional Boltzmann distribution, evaluate the generated distribution w.r.t. the target properties of the single targeted features and update their weights. By suitable updates of the weights, it is possible to shift the distribution towards the targeted feature values and increase the probability to satisfy these targets (within the given tolerance). During this entire learning procedure, Infrared returns samples inside of the tolerance range and rejects all others. In this way, Infrared implements a variant of multi-dimensional Boltzmann sampling [2], which let's it solve the kind of complex constraints set are set by targeting certain tolerance ranges for features (which are composed from 'local' functions).

As a consequence of this entire mechanism, the sampling efficiency in Infrared is a result of the complexity of the constraint network as well as the (in)dependence of the targeted features and the demanded tolerances. For practical applications of Infrared it is thus generally advantageous to be aware of the properties of the solving strategy and the resulting dependencies between these factors. This is especially important, since the framework easily allows modeling extremely hard problems, while (as we demonstrate) it is useful for a wide range of applications in practice. Its specific properties make the system attractive for a variety of complex design applications but as well intrinsically influence its applicability.

3.7 Negative design by direct sampling

Good RNA designs typically must satisfy certain constraints and show high (or specific) affinity to the target structures, but as well must avoid high affinity to all non-target structures. Objectives of the latter type are called negative design criteria (whereas the former are called positive design criteria). Intuitively, design towards negative criteria seems harder since it requires to avoid affinity to exponentially many structures. Indeed, as we discussed before, negative design was shown to be NP hard in relevant settings.

For our negative design examples, we make use of the Turner energy model and other functionality provided by the Vienna RNA package, which we interface via its Python module `RNA`. Note this is currently not supported for Windows, such that the negative design examples can not be run on Windows systems. Under Linux and Mac, we simply import the module by

```
import RNA
```

A classic (single-target) negative design objective, requires the target structure to have minimum free energy among all structures of the designed RNA, which can be tested using the Vienna RNA package:

```
def is_mfe_design(sequence, target):
    fc = RNA.fold_compound(sequence)
    return fc.eval_structure(target) == fc.mfe()[1]
```

In many cases, sampling (targeting only positive design criteria) can be sufficient to satisfy negative design criteria. For example, we can easily find designs for any of our three target structures by straightforward "generate-and-test":

```
sampler = Sampler(single_target_design_model(target))
sampler.set_target(0.7 * n, 0.1 * n, 'gc')
for i in range(50):
    seq = ass_to_seq(sampler.sample())
    if is_mfe_design(seq, target):
        print(f"{i} {seq}")
```

where `design_model(target)` returns a model for the given target structure with 'gc' feature and a bias to good base pair energy as we have developed it before (see Notes). Based on this code, we easily find 10 to 20 solutions from generating 50 (biased) samples for each of our example target structures (in less than 0.1 seconds on a current notebook).

A similar approach can still be sufficient to optimize other negative criteria for single-target design like the frequency of the target structure (see Notes for function `target_frequency`). The following code regularly finds designs with $> 80\%$ target frequency for our example targets in comparable run-time (roughly, twice as long).

```
sampler = Sampler(single_target_design_model(target))
sampler.set_target(0.7 * n, 0.1 * n, 'gc')
best = 0
for i in range(100):
    seq = ass_to_seq(sampler.sample())
    freq = target_frequency(seq, target)
    if freq > best:
        best = freq
        print(f"{i} {seq} {freq:.6f}")
```

Note that in both cases, we control the GC content of our designs. This is just one example how the core idea of finding designs by sampling can be extended by further targets or constraints. (Adding prior knowledge as IUPAC string,

would be another.) Generally, this approach will be most successful for problems with a medium-sized solution space. This would also apply to many design problems, where only parts of the RNA should be redesigned, while others are kept constant.

3.8 Larger single-target designs by constraint-generation

For larger and harder single-target design instances, we suggested a constraint-generation strategy in RNAPOND [13]. Here, we demonstrate a slightly stripped down version of this appraoch, which requires only a few lines of code using Infrared.

The method starts with the attempt to solve negative design by sampling (=positive design) as in the subsection before. When we fold the suggested designs, it can be observed that some base pairs that don't occur in the target, occur more frequently than others. This motivates us to identify these most frequent disruptive base pairs and forbid them in a next round of sampling. This strategy is iterated until a design (according to the mfe criterion) is discovered. In the code presented here, we decide to terminate the design attempt, when the problem becomes inconsistent or a certain complexity of the constraint model is exceeded.

```
def cg_design_iteration(dbps):
    model = single_target_design_model(target)
    model.add_constraints(NotBPComp(i, j) for (i, j) in dbps)
    sampler = Sampler(model, lazy=True)
    if sampler.treewidth() > 10 or not sampler.is_consistent():
        return "Not found"
    ctr = Counter()
    sol = None
    for i in range(100):
        seq = ass_to_seq(sampler.targeted_sample())
        fc = RNA.fold_compound(seq)
        mfe, mfe_e = fc.mfe()
        if fc.eval_structure(target) == mfe_e:
            sol = seq
        ctr.update(parse(mfe))
    ndbps = [x[0] for x in ctr.most_common() if x[0] not in bps]
    dbps.extend(ndbps[:2])
    return sol
dbps, seq = [], None
```

```
while seq is None: seq = cg_design_iteration()
print(seq)
```

We tested the code to find optimal 'mfe' designs for instances from the Eterna100 benchmark set [9]. Note that this code does not yet match the performance of the more refined tool RNAPOND, but it is still quite effective. For example, an optimal design for instance 39 ("Adenine") is typically found after 7–9 iterations.

Figure 4, provides intuition of the effect of adding disruptive base pair constraints in our strategy. Following the code above, RNA sequences are first sampled without further limitation than the target structure. Positions corresponding to the target structure have a greater chance to form a base pair according to the MFE criteria. However, as shown in the first plot in Figure 4, some unwanted base pairs are also observed considerably. As seen in the second plot, the two most frequent unwanted base pairs in the first round are restricted from paired (via constraint `NotBPComp`) in the second round of sampling. Then, additional constraints are imposed on two novel disruptive base pairs discovered in the following sampling round. After 7 rounds of sampling with 14 disruptive base pairs restricted, a deign sequence is successfully found, `CCGACAAGGGCCAUGCGCCCGGAAACCAGUGCCUCUGAAGUCAAAGUCUG`, for the target structure `"..(((..((((.....)))).((...(((.....)))...))...)))..."`.

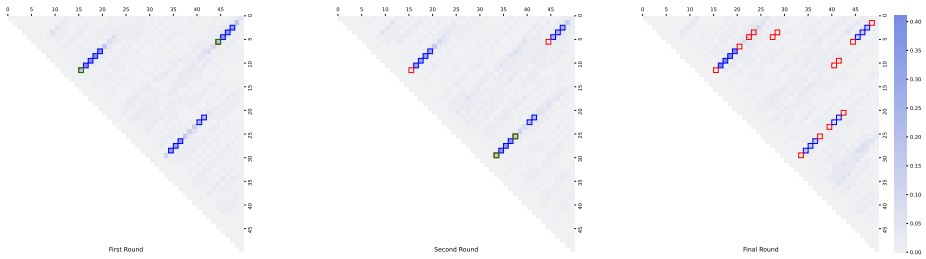

Fig. 4. Base pair frequencies and disruptive base pairs in the first, second and final iteration of a typical run of our RNAPOND-like constraint generation design strategy targeting `"..(((..((((.....)))).((...(((.....)))...))...)))..."`. In our triangular matrix visualization, each point refers to one potential base pair (i,j); base pair frequencies the generated sample are color coded and the target base pairs are highlighted by blue squares. By adding disruptive base pairs in each iteration (red squares), the frequencies of the base pairs are shifted towards the target ones.

3.9 Negative design by stochastic optimization with partial resampling

Infrared supports stochastic optimization strategies to optimize negative objectives for single or multi-target design. We are going to present a Metropolis-Hastings optimization strategy for optimizing the solutions from a multi-design models. It starts with a sample from the model and evaluates it by the objective function. Then, it iteratively picks a connected component of the dependency graph; constructs a model for resampling of the variables in this component; and generates a sample. Based on its evaluation by the objective function, the sample is either accepted or rejected based on the Metropolis-Hastings criterion.

For our example, we choose to minimize the multi-defect [6], which is a weighted sum of the distance of the target energies to the ensemble energy and the energy distance between the targets. Minimizing this function thus means to increase the probability of the targets and balance the target energies as much as possible (see function `multi_defect` in Notes).

Next, we define a function `multi_design_model` (see Notes) that returns a multi-target design network model (closely resembling the one presented before). In addition, crucially for the use in stochastic optimization, this model supports partial resampling. Given a subset of the model variables and a complete solution, it fixes all variables outside of the subset to the solution. This is accomplished by the code snippet

```
for i in range(n):
    if i not in subset:
        value = solution.values()[i]
        model.restrict_domains(i, (value, value))
```

In this function (see Notes), we moreover avoid to introduce constraints on or between variables outside of the subset.

For the stochastic optimization, we require additional imports

```
from random import random, choices
from math import exp
```

The function `multi_design_optimize` returns the best multi-target design and its multi-defect after a number of iterations (`steps`); a further parameter `temp` (temperature) controls the acceptance probability.

```
def multi_design_optimize(steps, temp):
    cc, cur, curval, bestval = None, None, math.inf, math.inf
    for i in range(steps):
```

```
        model = multi_design_model(cur, cc)
        new = Sampler(model).sample()
        newval = multi_defect(ass_to_seq(new), targets, 1)
        if (newval <= curval
            or random() <= exp(-(newval-curval)/temp)):
            cur, curval = new, newval
            if curval < bestval:
                best, bestval = cur, curval
        if i==0:
            ccs = model.connected_components()
            weights = [1/len(cc) for cc in ccs]
        cc = choices(ccs, weights)[0]
    return (ass_to_seq(best), bestval)
```

Finally, we run the multi-defect optimization on our example target structures by

```
multi_design_optimize(1000, 0.015)
```

Note that here the number of 1000 iterations and the temperature 0.015 were chosen after some experimentation. In practice, we will often restart such procedures to obtain better solutions and/or a diverse set of good solutions. In our tests, one such optimization run took moderate run-time below 10s on current notebook hardware, while it can find very good designs for the targets.

Figure 5 shows the best multi-defects in 48 runs after up to 5120 iterations. The best sequence `CCCUGUGCUCCAUGGGCCCCCGUCAGGGGACGGGG` that was found in these 48 runs of optimization had an multi-defect of 1.31. For this sequence, the target structures have respective energies -16.9, -16.5, and -16.9 kcal/mol; two of the targets have minimum free energy. This small experiment yields insights into the effectivity and convergence of the optimization procedure. For applications, it seems to suggest an optimization strategy combining restarts and moderately long runs.

Remarkably, there are even solutions where all three targets have minimum free energy. For the sequence `CCCCUUGCCUCAAGGGCCCUCUUCAGAGGAAGGGG`, which was discovered by the same strategy, all three target structures have a free energy of -15.40 kcal/mol (at a multi-defect of 1.21). Even the close suboptimals contain only structures similar to the targets as can be seen from the output of `RNAsubopt` of the Vienna RNA package, which enumerates all structures within 1 kcal/mol of the minimum free energy.

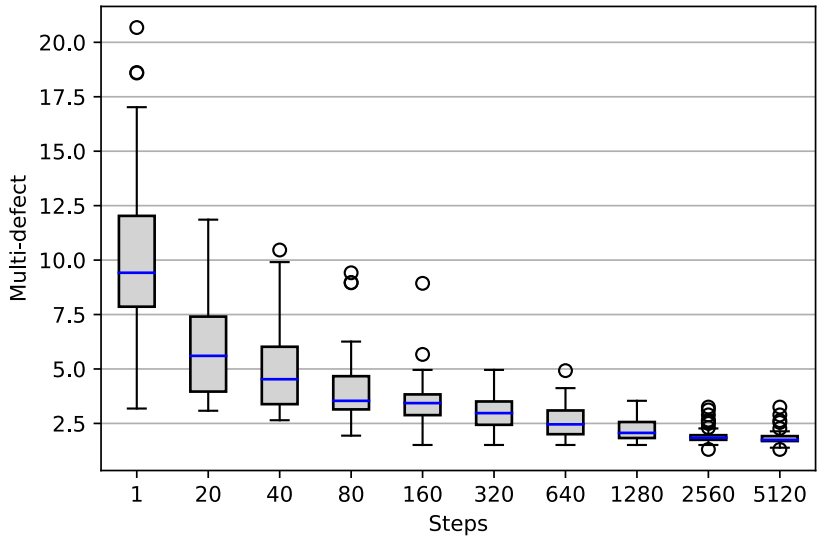

Fig. 5. Distributions of best multi-defects from 48 runs of `multi_design_optimization` without stochastic optimization (step 1) and after up to 5120 optimization steps at temperature 0.015. In the box plots, the boxes extend from quartile to quartile; the medians are shownn in blue; and the whiskers reach 1.5 times beyond the quartile. Remaining data points are shown as circles.

```
$ RNAsubopt -s <<<CCCCUUGCCUCAAGGGCCCUCUUCAGAGGAAGGGG
CCCCUUGCCUCAAGGGCCCUCUUCAGAGGAAGGGG -15.40   1.00
((((((((((...))))((((....))))))))))) -15.40
((((((.(((((((....))))..))))))))))) -15.40
.((((((...)))))).(((((((....))))))) -15.40
((((((.(((((((......)))..))))))))))) -15.10
((((((.((((.(((....)))...))))))))))) -14.80
((((((.((((.(((.....)))..))))))))))) -14.80
((((((.((((..(((....)))..))))))))))) -14.80
((((((.((((.((((...))))..))))))))))) -14.60
(((((((((.....)))((((....))))))))))) -14.50
((((((.((((.((......))...))))))))))) -14.50
.((((((...))))))(((.((((....)))))))) -14.50
```

3.10 A real world example: design of a Tandem-Riboswitch

Finally, inspired by collaborative work [5], we showcase the design of a tandem construct of two riboswitches for applications in bio-technology.

The single riboswitches control transcription by forming a terminator in their ground state, but they can as well bind a metabolite in an alternative aptamer structure. Upon binding, the terminator is destabilized and transcription is turned-on (Fig. 6). By combining riboswitches that react to two different metabolites (here Theophylline and Tetracycline) in tandem, we aim at the design of an AND-riboswitch, which ramps up gene expression (only) in the presence of both metabolites (Fig. 7)

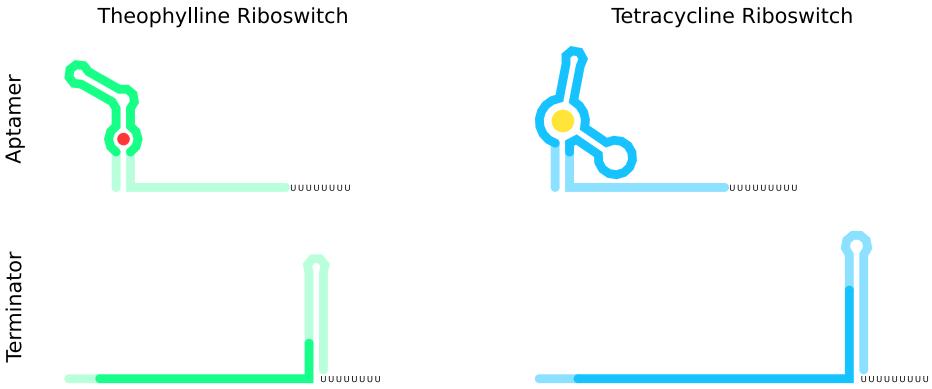

Fig. 6. States of a Theophylline (Theo) and a Tetracycline (Tet) riboswitch. For both riboswitches, we show the configuration that can bind the respective aptamer (top) and the competing terminator configuration, which terminates transcription (bottom). The regions with dark color represent functional sequences. On the other hand, the light one are the sequences that are free for design.

We use Infrared to suggest designs that connect two known aptamer constructs by a spacer region. Such design candidates could then be evaluated by biochemistry experts and tested in wet lab experiments (as e.g. done in [5]).

In preparation, we define the four structures of Fig 6 as dot-bracket strings. Moreover, we derive sequence strings from experimentally verified functional riboswitches, but replace some nucleotides by 'N' to leave additional freedom in the design (see definition of `seqTheo`, `aptTheo`, `termTheo`, `seqTet`, `aptTet`, `termTet` in Notes). These allow to compose the structural targets and sequence strings for the tandem construct, including a 30nt spacer (see Notes).

As goals for the computational design, we optimize the stability of the terminator structures (in absolute terms as well as in comparison to the structure ensemble), while keeping certain probabilities for the aptamer structure. Moreover, we want to avoid that the spacer region forms any stable structures or interferes with the structures of the riboswitch components. How to express such

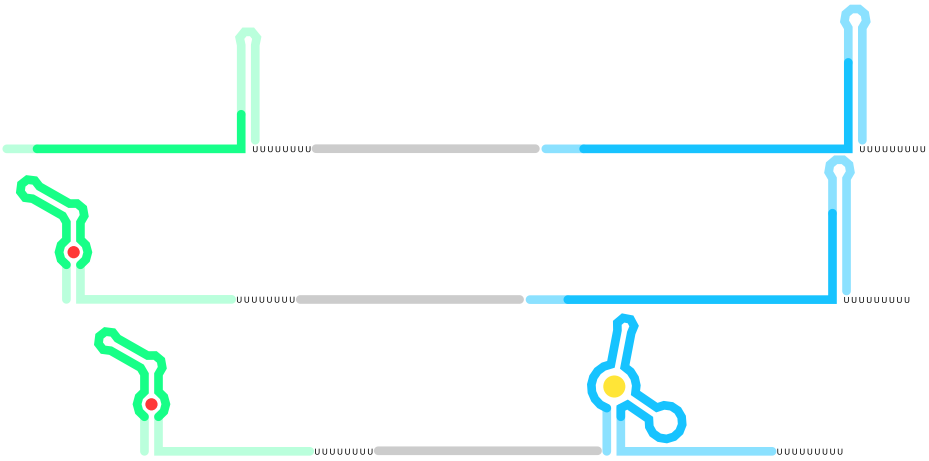

Fig. 7. States of an AND-riboswitch in the absence of Theo and Tet (top), in the presence of Theo and the absence of Tet (middle), and, in the presence of Theo and Tet (bottom). In our design example, we put Theophylline and Tetracycline riboswitches in tandem (Theo-Tet), connected by a freely designable 30nt space region (gray).

objectives as a function `rstd_objective` is described in details in our Notes. We make use of free energy differences between the free RNA structure ensemble (free ensemble energy) and constrained ensembles, which are constraints to form either aptamer or terminator structure, or keep the space unpaired.

It becomes clear, that we face a combination of positive and negative design goals under additional constraints. We suggest to perform this optimization as seen before by a stochastic optimization procedure using resampling of components in a constraint network. Consequently, to implement the entire design approach, we define a function `rstd_model` to set up a (resampling) model for the riboswitch tandem design and finally adapt the Metropolis-Hastings optimization scheme in `rstd_optimze`. Code for both functions is provided in Notes.

Finally, we can run the optimization for a specified number of steps and temperature:

```
rstd_optimize(steps = 500, temp = 0.03)
```

To find better and/or several design candidates, the optimization procedure can be repeated (or even run in parallel). For example, we generate designs like

```
CCGUGAUACCAGCAUCGUCUUGAUGCCCUUGGCAGCGCGGCCCGACAUCGGGCCGUGUUGUUUUUUUUCCC
↪ ACACACAAUAAAGAUAAACCUACCCGGCCGUUAAAACAUACCAGAGAAAUCUGGAGAGGUGAAGAAU
↪ ACGACCACCUAGCGGCUACCACCGUUAGGUGGUCGUUUUUUUUU
```

in several minutes—after 12 parallel runs. For this sequence, both terminators are very stable (energy differences of 0.14 and 0.0 kcal/mol), the stability of the aptamer structures is close to the target values (differences below 0.03 kcal/mol) and the designed spacer region behaves structurally almost neutral (low unfolding energy of 2.11 kcal/mol).

As demonstrated, Infrared appears technically well equipped to express various design constraints and target complex design objectives. Especially due to its declarative Python interface it moreover supports to apply flexible optimization strategies that make use of its capable design sampling engine. It turns out that this allows finding good solutions to practical design tasks using only 'a few lines' of code (which can still be discussed and printed in a bookchapter). This allows practitioners to focus on the remaining, 'true' challenges of computationally supporting the design of functional RNA molecules, as they undoubtedly exist in the realistic modeling of the design problem and the evaluation of promising design candidates that could be scrutinized in subsequent lab experiments.

4 Notes

4.1 A model for single target design

```
def single_target_design_model(target):
    n, bps = len(target), parse(target)
    model = Model(n, 4)
    model.add_constraints(BPComp(i, j) for (i, j) in bps)
    model.add_functions([GCCont(i) for i in range(n)], 'gc')
    model.add_functions([BPEnergy(i, j, (i-1, j+1) not in bps)
        for (i, j) in bps], 'energy')
    model.set_feature_weight(-1.5, 'energy')
    return model
```

4.2 Target frequency in ensemble

The frequency of the target structure in the ensemble of the designed sequence is a good example of a negative design criterion.

```
def target_frequency(sequence, target):
    fc = RNA.fold_compound(sequence)
    fc.pf()
    return fc.pr_structure(target)
```

4.3 Multi-Defect

The multi-defect can be computed with the help of the Vienna RNA library.

```
def multi_defect(sequence, targets, xi=1):
    k = len(targets)
    fc = RNA.fold_compound(sequence)
    ee = fc.pf()[1]
    eos = [fc.eval_structure(target) for target in targets]
    diff_ee = sum(1/k * eos[i] - ee for i in range(k))
    diff_targets = sum(2/(k*(k-1)) * abs(eos[i]-eos[j])
        for i in range(k) for j in range(k) if i < j)
    return diff_ee + xi * diff_targets
```

4.4 Multi-design model with support for partial resampling

```
def multi_design_model(subset=None, solution=None):
    n = len(targets[0])
    model = Model(n, 4)
    if subset is None: subset = set(range(n))
    for i in set(range(n)) - subset:
        value = solution.values()[i]
        model.restrict_domains(i, (value, value))
    model.add_functions([GCCont(i) for i in subset], 'gc')
    for target in targets:
        s = parse(target)
        ss = [(i, j) for (i, j) in s
                     if i in subset or j in subset]
        model.add_constraints(BPComp(i, j) for (i, j) in ss)
        model.add_functions([BPEnergy(i, j, (i-1, j+1) not in s)
                             for (i, j) in ss], 'energy')
    model.set_feature_weight(-1, 'energy')
    model.set_feature_weight(-0.3, 'gc')
    return model
```

4.5 Design of a tandem-riboswitch

Defining the design task starts with a definition of the two components of the tandem construct.

```
seqTheo  = "NNNNGAUACCAGCAUCGUCUUGAUGCCCUUGGCAGCNNNNNNNNNNNNNN"\
           "NNNNNNNNNNUUUUUUUU"
aptTheo  = "(((((...(((((((.....)))))...)))...)))))........."\
           "..................."
termTheo = ".............................(((((((((((....))"\
           "))))))))))......."

seqTet   = "NNNNNAAAACAUACCAGAGAAAUCUGGAGAGGUGAAGAAUACGACCACCU"\
           "ANNNNNNNNNNNNNNNNNNNNNNNNNNUUUUUUUUU"
termTet  = "......................................((((((((((""\
           "(((((......))))))))))))))........."
aptTet   = "((((((.......(((((....)))))...((((...........)))))"\
           ")))))............................."
```

These are concatenated while adding a spacer region, which are used to derive the model constraints.

```
spacerLen = 30

aptamers    = aptTheo  + "."*spacerLen + aptTet
terminators = termTheo + "."*spacerLen + termTet
sequence    = seqTheo  + "N"*spacerLen + seqTet
```

To express the objective function we define relevant structural variants of the entire tandem construct.

```
n = len(aptTheo) + spacerLen + len(aptTet)
variants = dict(
    empty = '.'*n,
    aptTheo = aptTheo + '.'*(n-len(aptTheo)),
    aptTet = '.'*(n-len(aptTet)) + aptTet,
    termTheo = termTheo + '.'*(n-len(aptTheo)),
    termTet = '.'*(n-len(aptTet)) + termTet,
```

```
    spacer = '.'*len(aptTheo) + 'x'*spacerLen + '.'*len(aptTet)
)
```

The riboswitch tandem design objective function consists of three terms:

- the stability of the two terminator structures, measured as free energy difference between the energies of the unconstrainted ensemble (empty constraint) and the ensembles that are constrained by the respective terminator structure. For each terminator, this difference has a natural interpretation as the required (additional) energy to form this terminator in the equilibrium state. Minimizing this energy targets maximal stability of the terminators.
- the difference between certain stability targets and analogous energy differences for the aptamer structures (7.0 kcal/mol for the Theo RS; 10.0 kcal/mol for the Tet RS). This targets specific unfolding energies for the aptamers. Minimizing the difference targets specific aptamer stabilities.
- the required energy to break any structures involving the spacer region. Minimizing this energy avoids interference of the spacer with other structure as well as stable structure within the spacer region.

```
def constrained_efe(sequence, c):
    fc = RNA.fold_compound(sequence)
    fc.hc_add_from_db(c)
    return fc.pf()[1]

def rstd_objective(sequence):
    efe = {k:constrained_efe(sequence, variants[k])
           for k in variants}
    term_stability = efe['termTheo'] + efe['termTet'] \
        - 2*efe['empty']
    apt_target = abs(efe['aptTheo'] - efe['empty'] - 7.0) \
        + abs(efe['aptTet'] - efe['empty'] - 10.0)
    spacer_unfolding = efe['spacer'] - efe['empty']
    return term_stability + apt_target + spacer_unfolding
```

The model sets up the constraints due to the sequence string and the base pairs in the aptamer and terminator structures. Moreover it sets weights for the features of base pair energies and GC content. The function is prepared to generate resampling models (which fix the sequence out of a specified subset of positions that are to be resampled) for its use in Metropolis-Hastings stochastic optimization.

```
def rstd_model(subset=None, solution=None):
    rstd_targets = [aptamers, terminators]
    n = len(rstd_targets[0])
    model = Model(n, 4)
    if subset is None: subset = set(range(n))
    for i in set(range(n)) - subset:
        value = solution.values()[i]
        model.restrict_domains(i, (value, value))
    for i, x in enumerate(sequence):
        model.add_constraints(ValueIn(i, iupacvalues(x)))
    model.add_functions([GCCont(i) for i in subset], 'gc')
    for k, target in enumerate(rstd_targets):
        s = parse(target)
        ss = [(i, j) for (i, j) in s
                    if i in subset or j in subset]
        model.add_constraints(BPComp(i, j) for (i, j) in ss)
        model.add_functions([BPEnergy(i, j, (i-1, j+1) not in s)
            for (i, j) in ss], f'energy{k}')
    model.set_feature_weight(-0.6, 'energy0')
    model.set_feature_weight(-1, 'energy1')
    model.set_feature_weight(-0.3, 'gc')
    return model
```

Finally, the objective function is minimized by a stochastic procedure (as seen before). We provide the adapted code:

```
def rstd_optimize(steps, temp):
    cc, cur, curval, bestval = None, None, math.inf, math.inf
    for i in range(steps):
        model = rstd_model(cc, cur)
        new = Sampler(model).sample()
        newval = rstd_objective(ass_to_seq(new))
        if (newval <= curval
            or random() <= exp(-(newval-curval)/temp)):
            cur, curval = new, newval
            if curval < bestval:
                best, bestval = cur, curval
```

```
        if i==0:
            ccs = model.connected_components()
            weights = [1/len(cc) for cc in ccs]
        cc = choices(ccs, weights)[0]
    return (ass_to_seq(best), bestval)
```

References

1. Mirela Andronescu, Anne Condon, Holger H. Hoos, David H. Mathews, and Kevin P. Murphy. Efficient parameter estimation for RNA secondary structure prediction. *Bioinformatics*, 23(13):i19–i28, Jul 2007.
2. Olivier Bodini and Yann Ponty. Multi-dimensional Boltzmann Sampling of Languages. *Discrete Mathematics and Theoretical Computer Science*, DMTCS Proceedings vol. AM, 21st International Meeting on Probabilistic, Combinatorial, and Asymptotic Methods in the Analysis of Algorithms (AofA'10):49–64, Jun 2010.
3. Kévin Darty, Alain Denise, and Yann Ponty. VARNA: Interactive drawing and editing of the RNA secondary structure. *Bioinformatics*, 25(15):1974–5, August 2009.
4. Rina Dechter. Tractable structures for constraint satisfaction problems. In *Handbook of Constraint Programming*, pages 209–244. Elsevier, 2006.
5. Gesine Domin, Sven Findeiß, Manja Wachsmuth, Sebastian Will, Peter F. Stadler, and Mario Mörl. Applicability of a computational design approach for synthetic riboswitches. *Nucleic Acids Res.*, 45(7):4108, Apr 2017.
6. Stefan Hammer, Birgit Tschiatschek, Christoph Flamm, Ivo L Hofacker, and Sven Findeiß. RNAblueprint: flexible multiple target nucleic acid sequence design. *Bioinformatics (Oxford, England)*, 33:2850–2858, September 2017.
7. Stefan Hammer, Wei Wang, Sebastian Will, and Yann Ponty. Fixed-parameter tractable sampling for RNA design with multiple target structures. *BMC bioinformatics*, 20(1):209, 2019.
8. Kalev Kask, Rina Dechter, Javier Larrosa, and Avi Dechter. Unifying tree decompositions for reasoning in graphical models. *Artif. Intell.*, 166(1-2):165–193, Aug 2005.
9. Jeehyung Lee, Wipapat Kladwang, Minjae Lee, Daniel Cantu, Martin Azizyan, Hanjoo Kim, Alex Limpaecher, Sungroh Yoon, Adrien Treuille, Rhiju Das, and EteRNA Participants. RNA design rules from a massive open laboratory. *Proceedings of the National Academy of Sciences of the United States of America*, 111:2122–2127, February 2014.
10. Ronny Lorenz, Stephan H Bernhart, Christian Höner Zu Siederdissen, Hakim Tafer, Christoph Flamm, Peter F Stadler, and Ivo L Hofacker. ViennaRNA package 2.0. *Algorithms for molecular biology : AMB*, 6:26, November 2011.
11. Vladimir Reinharz, Yann Ponty, and Jérôme Waldispühl. A weighted sampling algorithm for the design of RNA sequences with targeted secondary structure and nucleotide distribution. *Bioinformatics*, 29(13):i308–i315, 06 2013.
12. Douglas H Turner and David H Mathews. NNDB: the nearest neighbor parameter database for predicting stability of nucleic acid secondary structure. *Nucleic acids research*, 38(Database issue):D280–D282, January 2010.

13. Hua-Ting Yao, Jérôme Waldispühl, Yann Ponty, and Sebastian Will. Taming Disruptive Base Pairs to Reconcile Positive and Negative Structural Design of RNA. In *Research in Computational Molecular Biology - 25nd Annual International Conference, RECOMB 2021*, Apr 2021.