A New Algorithm for the Inverse Folding of RNA

Mirela Andronescu
University of British Columbia
Department of Computer Science
andrones@cs.ubc.ca

Firas Hamze
University of British Columbia
Department of Computer Science
fhamze@cs.ubc.ca

Frank Hutter
University of British Columbia
Department of Computer Science
hutter@cs.ubc.ca

Anthony P. Fejes
University of British Columbia
Department of Microbiology and Immunology
fejes@interchange.ubc.ca

April 16, 2002

Abstract

In this paper, we propose a new method of solving the RNA inverse folding problem, which includes a heuristic for initializing the sequence of bases and a Stochastic Local Search (SLS) algorithm for finding an optimal solution. The new algorithm is consistently able to solve structures for which the Vienna Package algorithm is unable to find solutions. The heuristic approach and the SLS optimization are also able to solve these structures significantly faster.

1 Introduction

The central dogma of molecular biology was originally described by Francis Crick and George Gamov in 1957 [1, 2] which described the relations between deoxyribose nucleic acids (DNA), ribose nucleic acids (RNA) and proteins as best it could be summarized at that time. As the research became better understood and the central dogma was fleshed out, it became apparent that the relations between these large biomolecules was much more complex than the simple transcription and translation scheme that had been originally proposed. DNA and proteins have been shown to interact in a great many ways that are vital to the functioning of the cell, and we now know that translation is carried out by an RNA-protein complex called a ribosome [3]. While the central dogma of molecular biology remains accepted as the canonical schematic of cell function, it has been supplemented to the point where it is simply an abstract formulation of a vastly more complex set of processes.

Even in the simplest version of the central dogma, RNA plays a number of roles. The exact transcript created from the DNA template is synthesized as messenger RNA (mRNA), the ribosome which translates the RNA message into proteins does so using ribosomal RNA (rRNA), and the carriers of the free amino acids are made of transfer RNA (tRNA). In more complex systems, short RNA fragments play a role in spliceosome recognition of splice sites, as well as essential functions including telomere extensions [4]. In these and a great number of other processes, RNA plays an integral role as a simple linear molecule, involved in pattern recognition. However, the secondary structure of both rRNA and tRNA plays a vital role in determining their function. What has become obvious for both proteins and RNA is that knowing their primary structure, which is their sequence, is insufficient to determine either their function or their role in the cell.

The problem of folding RNA molecules is considered less challenging than that of folding proteins. Although a great amount of work has been published on protein folding, often referred to as the "Holy Grail" of molecular biology [5], secondary structure remains a challenge and tertiary structure prediction still remains beyond our reach. Designing RNA secondary structures, however, can be relatively accessible, despite the challenges involved in predicting three dimensional conformation or tertiary structure. The two main challenges remaining to be solved include predicting pseudo-knots [6], which are the interactions between sub-elements of the fold, and the assembly of the secondary structure into its three dimensional form. Unlike protein folding, however, the number of RNA folds being solved is quite small and, in general, limited to a small set of ribosomal RNA molecules.

Despite the current relative lack of work in the field of RNA structure prediction, research into catalytic RNA molecules, termed rybozymes, may prompt a greater interest in the study of RNA molecules [7, 8]. Rybozymes, which have the ability to catalyze cleavage of other RNA molecules, have been demonstrated in a number of systems and have broadened the modern view of the role of RNA outside the central dogma of molecular biology [9]. It has been hypothesized that catalytic RNA molecules may have played a greater role in early evolution, before the complex proteins were evolved. As well, it has also been suggested that rybozymes may provide novel paths to the design of new drugs [10, 11, 12] or active molecules which may find an industrial use [13]. These applications may also help to broaden the current interest in RNA molecules and their folding.

One of the challenges that will certainly be faced in the future in the laboratory setting is the "Inverse RNA folding" problem [14] discussed here. Rather than predicting the structure for a given sequence, the more challenging problem of finding a sequence for a given structure is tackled. Among the many difficulties in solving this problem is that a single given structure may be solved by a great number of possible sequences. Like protein folding, it is possible for multiple distinctly different sequences to yield the same final structure, once folded.

With a great number of potential future applications for designed catalytic RNA molecules, there is much to be gained by the study of inverse RNA folding.

2 RNA Secondary Structure and Notational Conventions

Simple Watson-Crick basepairing involves the hydrogen bonding of either cytosine (C) and guanine (G) or adenine (A) and uracil (U), forming three or two hydrogen bonds respectively. However, in addition to these two possible pairings, a third type of pairing between guanine and adenine (G-A), not predicted by the Watson-Crick or Chargaff rules, can be observed. This unusual pairing is termed a "wobble" pair and is thermodynamically less stable than either of the cannonical Watson-Crick pairings. Base pairs are most often found stacked onto other base pairs in substructures called *stems* or *helices.* Sometimes, free bases are interspersed in stems; these are known as *internal loops* or *bulges.* Loops occurring at the ends of stems are called *hairpins*, and loops from which more that 2 stems originate are known as *multi-branched loops*, or simply *multiloops.* An intuition as to how these substructures form a secondary structure can be gained from Figure 1.

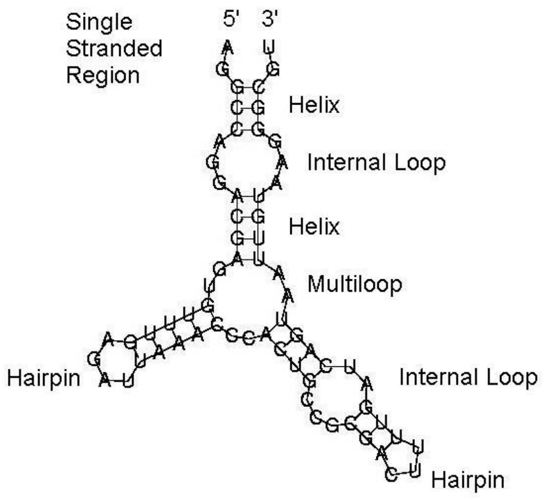

Figure 1: An example RNA structure showing a variety of features that are commonly found

The RNA structures we consider can be represented as nested strings. Each base in an RNA sequence is either a member of a *single-stranded* subsequence, that is, unpaired with another base, or of a *base-paired subsequence,* which forms hydrogen bonds with a base that is not adjacent to it in the sequence. However, because of the difference in the number of hydrogen bonds formed, C-G pairings (where C-G denotes hydrogen bonding between two non-adjacent bases) are able to form the most thermodynamically stable helices, while A and U are more frequently found as non hydrogen bonded free bases.

A convenient representation of secondary structure that captures the above features is known as *dot-and-parenthesis* notation. In Figure 1, an example RNA structure is shown, with the bases that constitute it denoted at each location. Note the presence of stacked pair regions, internal loops, a multiloop and single-stranded regions. Figure 2 shows the equivalent notation for the structure. Each open-closed parenthesis pair represents a base pair in a nested fashion; the dots signify the unpaired bases. Hence, regions of open parentheses correspond to helices, regions of dots between open and closed parentheses to hairpins, and areas where the parentheses switch from closed to open with remaining unpaired open parentheses signify multi-loops.

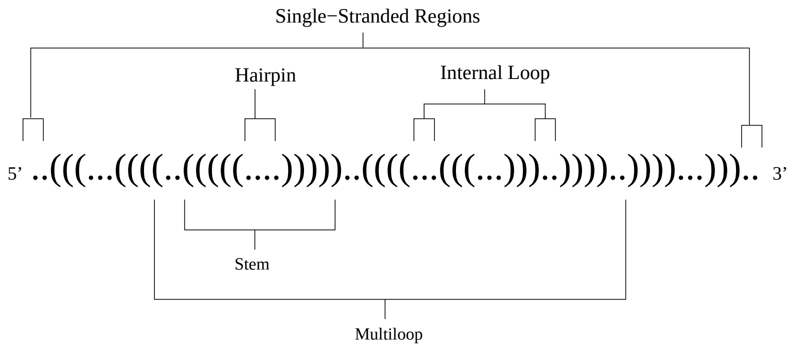

Figure 2: The dot-and-parentheses notation for the structure given in Figure 1. Some representative structural features are also shown.

3 RNA Folding and Inverse Folding: Definition and Approaches

After we motivated the inverse RNA folding problem in the previous sections, we now move on to a more algorithmic point of view on how to solve the problem. For this purpose, we start by defining the forward folding problem and the problem of inverse RNA folding (IRNAF) and then give an overview of current algorithms for these problems.

Let $E(x, \Omega)$ denote the free energy of an RNA sequence x when folded into the secondary structure Ω with respect to a given model to evaluate the energy. Furthermore, let Φ denote a function that assigns to each RNA sequence x the secondary structure Ω in which x folds with minimal energy $E(x, \Omega)$.

The RNA folding problem is the following: Given an RNA sequence x, determine $\Phi(x)$. Analogue, the inverse RNA folding problem can be defined as follows: Given an RNA secondary structure Ω^*, find a sequence x^* s.t. $\Phi(x^*) = \Omega^*$. In the optimization variant of this problem, we determine the quality of a candidate solution $\hat{x}$ by comparison of the structure $\hat{\Omega} = \Phi(\hat{x})$ with the desired structure Ω^*; given a distance metric d, we attempt to minimize $d(\hat{\Omega}, \Omega^*)$. In our algorithm, the distance metric will merely measure the number of bases that bond incorrectly; thus, $d(\hat{\Omega}, \Omega^*)$ is the number of bases in $\hat{\Omega}$ that bond differently than in Ω^*. It is also possible to include information on the stability of folds in the distance metric.

3.1 RNA Folding algorithms

When the energy $E(x,\Omega)$ of an RNA structure x folded into the secondary structure Ω is evaluated by the nearest neighbor thermodynamical model [24], the RNA folding problem is efficiently solvable using a dynamic programming approach. Zuker's optimal algorithm [15] ignores pseudo-knots and runs in time $O(n^3)$ with space requirements of $O(n^2)$. Rivas and Eddy propose an algorithm that runs in time $O(n^6)$ and handles restricted types of pseudo-knots [20].

3.2 Inverse RNA Folding algorithms

This section outlines a prominent inverse RNA heuristic by Hofacker et al. [16], which is included in the Vienna RNA Secondary Structure package [25]; we will often refer to this approach as the *Vienna inverse folding algorithm* or short *Vienna algorithm.* According to our best knowledge, the Vienna algorithm is the only method that has been proposed to algorithmicly solve the inverse RNA folding problem. The algorithm performs an "adaptive" walk search on so-called *compatible* sequences, i.e. segments that can possibly form a base-pair at the required positions in the desired structure. These sequences are candidates only; there is no guarantee that they will fold into the target structure. The approach is to begin from a compatible sequence x_0, randomly induce a mutation (while ensuring that the sequence remains compatible) and accept it if and only if the cost function decreases. This process is iterated until either a solution is found or a certain number of mutations has been carried out. The trouble with this approach is that calculating the distance for each mutation involves running a folding algorithm on the sequence under observation, for which generally Zuker's $O(n^3)$ algorithm is used, where n is the length of the sequence to be folded. To counter this, Hofacker et al. propose that individual substructures' subsequences be determined first and the final sequence be the one corresponding to a simple concatenation of the subsequences. The rationale is that it is likely (but not assured) that the substructures which are optimal for subsequences will also occur for the full sequence.

4 The Inverse RNA Folding algorithm (IRNAF) - Motivation

In this section, we introduce a new algorithm for the inverse RNA folding problem. The algorithm performs a Stochastic Local Search on substructures and combines partial sequences to a complete solution. We start by motivating the Stochastic Local Search (SLS) and state the problems occuring in a naive approach. We also give intuition on how to initialize the SLS in a more promising way than at random.

4.1 Stochastic Local Search

Stochastic Local Search (SLS) is a popular recent field of research in computer science. Its main objective is to solve hard combinatorial decision and optimization problems [22]. SLS algorithms are randomized algorithms that perform local search in a given search space, the search space consisting of *candidate solutions*; candidate solutions consist of a complete assignment to the solution components of the problem. In the case of inverse RNA folding, each base x_i in a sequence $x_1, \ldots, x_n$ is a solution component; the sequence x comprises a candidate solution. In a candidate solution, each of the bases x_i is assigned one of the four bases {A,C,G,U}; the bases that have to pair in the desired structure have complementary bases, i.e. C-G or A-U.

A straightforward local search approach for the inverse RNA folding problem starts by initializing the assignment of bases to the positions in a sequence. After this initialization, a series of reassignments to particular bases is performed (which we will refer to as the *flip* of a base), always preserving a candidate solution. A very basic algorithm, known as iterative improvement, executes a fixed number of flips and, if not having found a solution restarts with a new initialization.

Given this background, we can view the approach described in Section 3.2 as a simple SLS algorithm: after a random initialization, greedy base flips are executed, evaluating the possibilities of flipping a base and choosing the first one that improves on the current distance. [1]

Sometimes, it can be advantageous to do a random step (*noise*), regardless of how this changes the evaluation function value. Much research has been conducted in the role of noise in SLS, e.g. for Satisfiability [26, 27] and Constraint Satisfaction problems [28]. The – on the first glance surprising – result is that quite often high noise ratios around 50 percent result in the most stable algorithms. Since the Vienna algorithm does not include any noise there seems to be a great opportunity for improvement.

However, a simple first improvement strategy will not do a good job on its own. This is due to the extraordinary expensive evaluation of candidate solutions. At this time there is no faster available library for predicting the native structure of a given sequence more efficiently than the fold function from the Vienna Package (which implements Zuker's $O(n^3)$ algorithm [15]). The inherent difficulty in this procedure occurs because a single local reassignment of a base in the sequence can result in a completely different structure. Since sometimes SLS algorithms need a lot of (usually reasonably cheap) flips, there is little hope for SLS methods to work on large problem instances when the cost per evaluation scales cubic with the input size.

This motivates a hierarchical approach similar to the one described in Section 3.2, in which a structure is split into substructures, which are solved and merged together, hoping for a

[1] When having a superficial look at the actual implementation of the Vienna algorithm, it appeared that the possibilities of flipping a base in the sequence $x = x_1, \ldots, x_n$ are simply tried from x_1 to x_n. Since randomization often leads to superior algorithms [29], we assume that a randomized order would improve much on this; however, we did not try this due to our infamiliarity with the code.

minimal number of conflicts. As we will see later, this makes the algorithm perform much better on longer structures. Next, we give further intuition on the initialization of the SLS algorithm. Experiments to be described later have shown that a good initialization helps to achieve higher performance for various lengths of structures.

4.2 Intuitions for the Initialization Heuristic

In Section 2, we have mentioned the thermodynamic difference in base pairing energy between C-G pairs and A-U pairs. This provides the naive observation that a structure in which helices are comprised of only C-G pairs and hairpins and bulges made of A-T pairs would be the most stable, which can be measured by the Gibbs free energy of the system. However, it might be difficult to guarantee that the helices and loops will not find alternate conformations with a lower energy value. A heuristic initialization which assigns bases to generate a specific structure may benefit from assigning bases according to a certain set of probabilities, depending on the function of the region. Guided by thermodynamic stability conferred by the greater hydrogen bonding capacity of the C-C pairing as opposed to that of the A-U pairs, it makes sense to assign a high proportion of G's and C's to helices, and a lower proportion of A's and U's. In contrast, loops would benefit from the opposite arrangement. This provides the further advantage that the energy-minimized structure (generated by Zuker [15]) is unlikely to find structures in which segments intended to be helices bind to segments intended to be loops. Taking into account the pairing nature of the bases in helices, it is obvious that by assigning a sequence to the portions of the molecule that form helices, we are able to assign twice as many positions for half the work (Chargaff's rule.) In contrast, assigning bases to the loops presents a challenge because of the potential for the slippage of pairing bases in the adjacent helices. This can be prevented by altering the structures of helices as well as the bases adjacent to the last pair of the helix.

These observations make a logical starting point for designing structures. By assigning the positions in helices based on a statistical model, we can then skew the assignment of the nucleotides in favor of the Cs and Gs, but allowing the incorporation of A's and U's to introduce a greater variability. The issue of variability is paramount, as insufficient variety in sequences generated will contribute to unexpected energy minimized folds. As a check in the assignment of bases, it might be necessary to ensure that, at some level, assigned sequences are not repeated at other points in the RNA molecule. This is a quality control step that has also been used as the bases of the Vienna algorithm [16].

5 The Inverse RNA Folding algorithm (IRNAF) - Implementation

This section gives the actual implementation of the IRNAF algorithm; the outline of the algorithm is described in Figures 3 and 4:

While a given cutoff time is not reached do:

1. Initialize the sequence using the heuristic algorithm, described in Section 5.1;
2. Recursively split the given structure in two parts until it can no longer be split. The splitting algorithm is described in Section 5.2;
3. Apply a stochastic local search algorithm on each substructure that can no longer be split. This algorithm is described in Section 5.3;
4. Rejoin the subsequences and evaluate the result. If there is a conflict, go back into the recursion and try to resolve it. If the distance is not 0 after a fixed number of conflict repair tries, go back to step 1.

Output the best sequence found so far.

```
procedure InverseRNAFolder
    input: options, structure
    output: sequence
    sequence = InitializeSequence (options, structure);
    sequence = RNAInverse (options, structure, sequence, $\emptyset$);
    return sequence;
end InverseRNAFolder.
```

Figure 3: Pseudocode for the inverse folding algorithm

A variable called *options* is used to wrap parameters that are supplied to the algorithm. These parameters are as follows:

- A set of complimentary probabilities, *probCG* and *probAU*, are used to randomly assign values to nucleotides. These two parameters are used in the initialization phase as well as in the stochastic local search procedure when bases are flipped;
- A *terminate_split* parameter that contains the maximum number of potential attempts to split and join a substructure before an unsuccessful end of search is reported;
- A "choose probability" parameter p_c - used in the stochastic local search procedure;
- An "acceptance probability" parameter p_a - used in the stochastic local search procedure;
- A *terminate_sls* parameter, that contains the maximum number of steps the SLS procedure continues searching.

```
procedure RNAInverse
    input: options, structure, sequence, forbidden
    output: sequence
    if (structure cannot be split)
        return StochasticLocalSearch (options, structure, sequence);
    end if;
    (substructure1, substructure2) = HierarchicalSplit (structure);
    (subsequence1, subsequence2) = subsequences of sequence that correspond to
                                   (substructure1, substructure2)
    while (distance is not 0 and not terminate_split)
        forbidden = bases in sequence that bond incorrectly
        (forbidden1, forbidden2) = parts of forbidden that correspond to
                                   (substructure1, substructure2)
        subsequence1 = RNAInverse (options, substructure1, subsequence1);
        subsequence2 = RNAInverse (options, substructure2, subsequence2);
        create sequence from subsequence1 and subsequence2;
        distance = evaluate (sequence);
    end while;
    return sequence;
end RNAInverse.
```

Figure 4: Pseudocode for the algorithm of the recursive function

The *evaluate* (*sequence*) function is a call to the forward folding algorithm from the Vienna package, whose time complexity is $O(n^3)$. Note that this is the most costly function of our algorithm. This is why we call this function only once at this level, when we re-assemble the two pieces. If the distance of the sequence is not 0 and the parameter *terminate_split* still allows for another attempt, we split the structure again and look for a new local minimum, forbidding the assignment to bases that just resulted in an undesired bonding. The *evaluate* function is identical to the *evaluate* function called in the *StochasticLocalSearch* procedure, detailed in section5.3.

5.1 Initialization Phase

The first step is to split the structure into substructures of lengths not exceeding 10 bases, such that all elements in the substructure are identical (i.e. either all left brackets, right brackets or dots). A pool of accepted subsequences is created, which is empty before any of the substructures have been assigned bases.

Each substructure is then assigned random bases, by using the probabilities in the parameters *probCG* and *probAU*, which take into account the different probabilistic models for each type of substructure:

- greater probabilities for C and G and lesser probabilities for A and U for paired bases: parameter *probCG*;

```
procedure InitializeSequence
    input: options, structure
    output: sequence
    create substructures setsubs from structure;
    foreach substructure subs in the set setsubs that is of type left brackets
        using probability model probCG, generate a random sequence randseq for subs;
        add the complementary sequence of randseq in the pool;
        foreach sequence poolseq in the pool
            if (randseq and poolseq are complementary in more than k consecutive bases)
                remove randseq from the pool and start a new search for subs;
            else
                assign the complementary sequence to the corresponding bases;
                add randseq in the pool;
            end if;
        end foreach;
    end foreach;
    foreach substructure subs in the set setsubs that is of type dots
        using probability model probAU, generate a random sequence randseq such that
                the dots that are next to closing pairs of helices do not pair;
        add the complementary sequence of randseq in the pool;
        foreach sequence poolseq in the pool
            if (randseq and poolseq are complementary in more than k consecutive bases)
                remove randseq from the pool and start a new search for subs;
            end if;
        end foreach;
    end foreach;
    create sequence from the generated sequences;
    return sequence;
end InitializeSequence.
```

Figure 5: Outline of the heuristic algorithm that initializes a sequence given the desired structure

- lesser probabilities for C and G and greater probabilities for A and U for free bases: parameter *probAU*.

If these parameters are used in the opposite orientation, *probCG* for free bases and *probAU* for helices, the likelihood that of the structure generated matching the intended sequence is reduced. This is discussed in Section 5.

Once a substructure with a subsequence *randseq* is added to the pool, the complement of *randseq* is also added.

Each time bases are assigned to a substructure composed of left brackets, the complementary bases are assigned to the corresponding right brackets, and the sequence assigned, *randseq*, is added to the pool.

randseq is then compared to the subsequences in the pool to find if it can pair with more than a constant number k bases. If they do, the last added sequence is removed from the pool, another sequence *randseq* is generated and the procedure is repeated. If the new *randseq* does not form k pairs, *randseq* is considered good and the algorithm can proceed to the next substructure. The complement of *randseq* is included into the pool before checking in order to detect if there are conflicts within the same subsequence. The possibility of such conflicts may increase for long subsequences of dots. For example, if a substructure is .... and we do not add it into the pool first, the algorithm might generate a sequence that folds into ..(((((...))))).…... Adding *randseq* into the pool before looking for pairing sequences eliminates some of the potential for obtaining sequences that would behave unexpectedly.

Each time bases are assigned to a substructure composed of dots, we make sure that the dots next to closing pairs of helices can not form a pair. For example, if the structure is ((((.......)))), the substructures ((((, ... and)))) are identified. If the first dot and the last dot of the second substructure are able to pair, then the whole sequence will fold into (((((.....))))), instead of the desired structure.

The constant k must be large enough to allow for sufficient number of combinations to be used, as small values for k will toss out many more valid sequences. Initially, we used a value of 5 for k, but there is likely to be room for improvement (see section 6).

The pseudocode of the initialization algorithm is outlined in Figure 5.

5.2 Hierarchical Splitting

5.2.1 Overview

This section describes the methods used to divide an RNA structure into substructures. The motivation to do this is straightforward. As we shall see our algorithm's search phase relies heavily on computing the Vienna RNA [16] forward-folding function to determine the structure of a hypothesized sequence and observe how "far" the structure is from the desired one. The folding function is quite computationally expensive: it has time complexity $O(n^3)$. Any method that can reduce the lengths of the structures that the folding algorithm must operate on, while simultaneously keeping the amount of searching to be performed to a minimum, would be of great value to our system. In the following, we describe a previous method used to perform this subdivision, after which we give the details of our scheme.

Working with the intuition that helices occuring stably on their own are likely (but not guaranteed) to be preserved when joined to other helices at a multiloop, Hofacker et al [16] observe that it makes sense to divide the structure at the multiloops, perform the sequence search on the substructures, concatenate the 2 sequences derived, and perform any additional "fine tuning" that may be required (or reiterate the search phase a certain number of times).

The question remains, however, as to how to choose the divisions. A structure that contains many multiloops will yield many substructures emanating from the loops which can be combined in a number of ways. To clarify this, we note from [16] that the structure is representable as an ordered, rooted tree. In Figure 6 we show the equivalent tree for the structure shown in Figure 1. White and black nodes correspond to unpaired and paired bases respectively; the nodes can contain the position numbers in the original structure or the bases in the sequence (the black nodes contain 2 values, of course.) Performing a pre-order traversal of this tree returns the structural representation or the sequence, depending on what the nodes contain. Clearly, multiloops are areas where more than 1 black node originates from a parent.

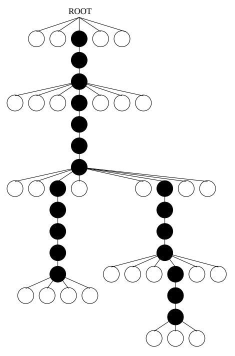

Figure 6: The tree representation of the structure shown in Figure 1. Free bases and paired bases are shown as white and black nodes respectively. Note the location of the multiloop, with 2 black nodes originating from a common parent. Adapted from Hofacker et al. [16]

One option for dividing the structure, as Hofacker et al [16] have done is to begin with the hairpins (i.e, the leaves of the tree where no black nodes are neighboring), to iteratively add base pairs and perform the sequence search, to concatenate the hairpins at multiloops, and to optimize the remaining base pairs connecting multiloops. The disadvantage is that as small substructures are added, the $O(n^3)$ forward-folding costs begin to increase. Towards the end of the search, if i small substructures are to be concatenated to a large substructure of length slightly less that N, (where N is the length of the full structure), then each time a substructure is added and the folding function is run, then slightly less the $O(N^3)$ computations are needed. For i "small" substructures, the complexity could approach $O(iN^3)$. The slight addition in length is unlikely to affect the stochastic search performance, and so the bottleneck will be the forward folding. Furthermore, if no solution is found on a large, concatenated substructure, backtracking and searching through the (slightly) smaller substructure can also be time-consuming. To circumvent these problems, we chose to subdivide the structure into two substructures of as close to equal lengths as possible, subject to the constraint that the split take place at a multiloop. In the tree representation, this corresponds to *finding the base pair node at a multiloop such that the subtree with that node as root has roughly equal length to the original subtree with that node and its subtree removed.* Note that this procedure can be repeated on the two generated subtrees if they also contain multiloops. The effect of this division of the structure displayed in Figure 1 is shown in tree representation in Figure 7. In Figure 8, we show a larger RNA structure and the two substructures it has been divided into.

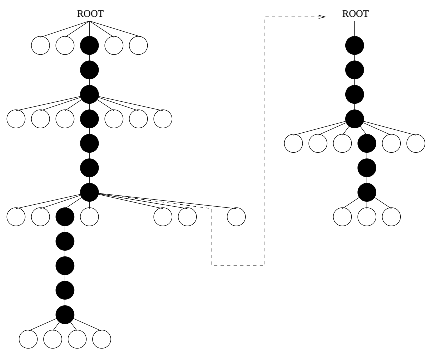

Figure 7: The tree representation for the 2 substructures resulting from splitting the one shown in Figure 1 and whose tree is displayed in Figure 6.

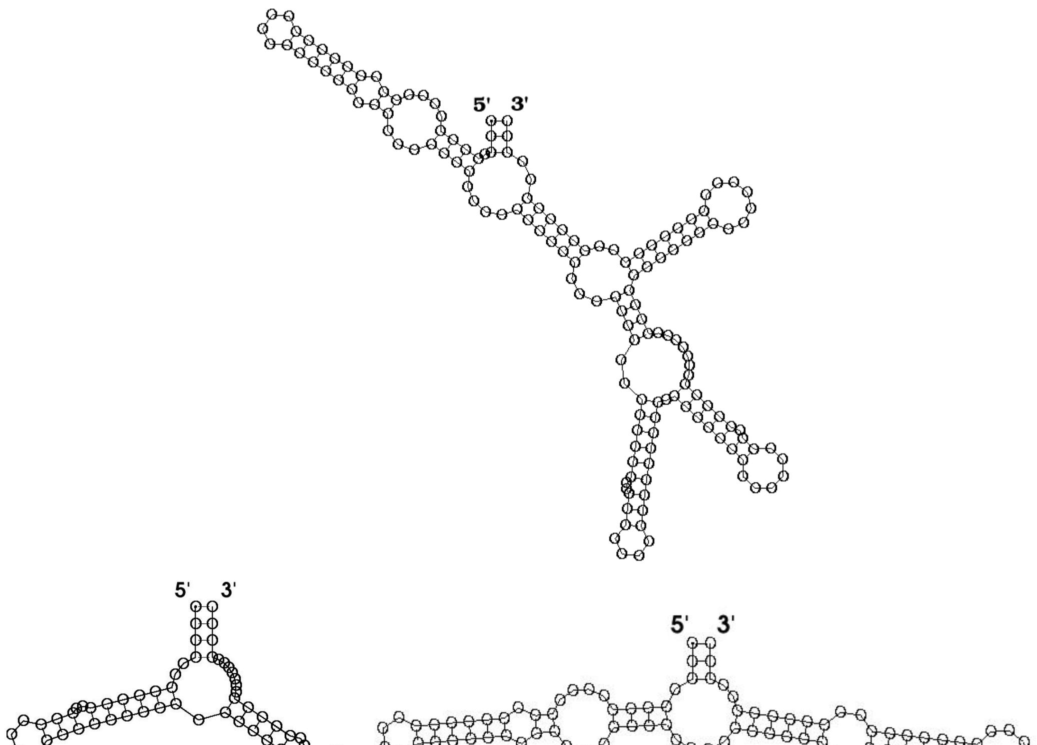

Figure 8: An example of a structure divided into two substructures using the method outlined in the text.

The pseudocode for the hierachical splitting procedure is shown in Figure 9. The idea is to first generate the tree representation of the structure and then find the node at a multiloop that divides the structure most evenly.

One technical difference between the tree we described in the previous section and the one we implemented is that instead of having each base be represented by a node in the tree, each *substructure* is contained in a node. The children of a substructure are those stems that are nested within it; to conserve the ordering the postions of the bases in the original sequence were stored along with the structure elements. Thus, the tree was equivalent to the one described. For example, a multiloop with 5 stems radiating from it would have 4 children: the parent node is the substructure of the stem entering (from the 5' end) and leaving (to the 3' end) the loop.

```
procedure HierarchicalSplit
    input: structure
    output: substructure1, substructure1
    Part I: Create the tree
    while (not terminate_create_tree)
        if (current_substructure_encountered) is a multiloop
            push current_substructure onto stack;
            push current_working_postions onto stack;
            New current_substructure;
        else
            append element from structure to current_substructure->string_linked_list;
            append position number in structure to current_substructure->position_linked_list;
        end if;
        if traversed_substructure and stack is not empty
            pop old_substructure from stack;
            increment old_substructure->number_children;
            old_substructure->children[number_children] = current_substructure;
        else if traversed_substructure and stack is empty
            terminate_create_tree;
        else
            parse next forward and backward position of structure;
        end if;
    Part II: Find the most close to equal-sized subtrees
    $mid\_node = argmin_{node}||length(node) - length(tree)/2||$;
    $substructure1 = subtree(mid\_node)$;
    $substructure2 = tree - subtree(mid\_node)$;
    return (substructure1, substructure1);
end HierarchicalSplit.
```

Figure 9: Pseudocode for the hierarchical split algorithm

Two further modifications to the splitting heuristic were also tried. We discovered after running the search on the divided structures that sometimes there was no sequence that could fold to the desired substructure. In other words, when they were part of the original structure, they were viable, but not on their own, or vice versa (see 2. below). Two such cases were found to be especially prominent:

1. Structures that contained a free base directly after the first parenthesis when it was the first substructure element. For example: **(.((((....)))))**

2. Structures that, as an artifact of the splitting procedure, contained "hairpins." The issue arose when the other substructures were re-inserted to create the original structure, and the free bases stacked with the base pairs located at the stem's end rather than remain free. In the following examples, we insert bars (—) in the structural representation as visual aids to show the relevant locations of insertion. The following example element of a substructure:
   ((((... | .....))))
   which had the following stem inserted:
   (((((...)))))
   should have yielded:
   ((((... | (((((...))))) | .....))))
   but instead, the forward folding algorithm would often fold the concatenated sequence to:
   ((((..(| (((((...))))) |)....))))
   In other words, the free bases which were stable on their own in the substructure would form stacked pairs in the concatenated structure.

Our solution to the first problem was to prefix and append the substructure with free bases, so that our example would become:
.. | (.((((....)))) | ..
We hypothesize that this remedies the situation because it now creates a target that is in fact realistic relative to the energy models used by the Vienna forward-folding algorithm.

The second problem was fixed by artificially inserting a stem between free bases that have been determined to have been split. This is found by checking whether their position numbers in the original structure are consecutive in the substructure. If not, a split was performed, and a stem of length 3 with a three base hairpin is attached. Thus, our substructure example is augmented to:
((((... | (((...))) | .....))))
the optimization is performed, and when concatenated to the other stem, the artificial hairpin is discarded. Choosing to insert this "3-3-3" helix was an intuitively reasonable way to force force "boundary conditions" similar to those that were found in the original structure while maintaining a short length. An alternative would have been to take, a stem of a certain length from the original structure, for example all the bases contained up to and including the third paired base leaving the stem, and attaching a hairpin to the end. Our preliminary experiments showed that this offered no clear advantage over the generic structure that we insert.

5.3 Stochastic Local Search on Small Substructures

Here, we give a Stochastic Local Search algorithm for substructures of the original structure. Since structures might still be of size around 100, and $100^3 = 1000000$, it is of high importance to keep the number of objective function evaluations of long sequences as low as possible. This leads to the first improvement strategy depicted in figure 10. The initial assignment of bases is locally changed in order to reduce conflicts. A flip of a base that should pair in the structure implicitly flips the base it pairs with, such that the pairing is preserved. If a base does not bond like in the desired structure, it is said to be *in conflict*.

```
procedure StochasticLocalSearch
    input: options, structure, sequence, forbidden
    output: sequence, an RNA sequence with native structure
                identical or similar to structure
                and sequence$_i$ $\neq$ forbidden$_i$
    while (not terminate_sls)
        do
            with probability $p_c$: $i \leftarrow$ index of a random base.
            otherwise: $i \leftarrow$ index of a random base that bonds incorrectly.
            $x_i$ = flip($i$,probCG, probAU);
        while ($x_i$ = forbidden$_i$);
        distance = evaluate (sequence);
        if (distance is better or equal than before the flip)
            sequence$_i$ $\leftarrow x_i$;
        else with probability $p_a$: sequence$_i$ $\leftarrow x_i$;
    end while;
    return sequence
end StochasticLocalSearch.
```

Figure 10: Stochastic Local Search for substructures

With a predefined *choose probability* p_c, a random base is chosen to be flipped. Otherwise, one of the bases in conflict is flipped randomly. If the result of this flip is a forbidden value for the base, the flip is undone and a new flip is tried. When the value is allowed, this flip is evaluated to determine if it results in a better structure than before (note, that this implies a call of Zuker's $O(n^3)$ algorithm). The flip is accepted if the resulting structure does not have a higher distance from the intended structure than the original one. In addition, worsening moves are accepted with an *acceptance probability* p_a.

The process of flipping bases is iterated until a predefined number of flips has been performed, or an optimal sequence x^* has been found.

5.4 Random Structure Generator

A random structure generator was designed in order to create simple artificial structures of any size that do not contain pseudoknots. By combining the simple elements that make up bracket notation, (,) and **.** in a rational manner, an unambiguous and complex structure can be created. Based on the simple rules upon which RNA folding is based, the complexity and size of generated structures can range from extremely simple and small to highly nested and large sequences. This flexibility allows a very wide variety of challenges to be presented to the folding algorithms.

The generated structures are composed of four separate elements:

1. Free Bases, represented by **.** may be added anywhere.
2. Helices, represented by $(_{\mathbf{m}})_{\mathbf{m}}$, to which multiloops or hairpins can be attached.
3. Multiloops, represented by $._{\mathbf{n}}$, to which helices can be attached.
4. Hairpins, represented by $._{\mathbf{l}}$, to which no further elements are attached.

These elements are sufficient to generate any potential structure that does not contain pseudoknots. However we have added the further restriction that no free base may be added where it would cause the structure **.(.** or **.).** to be created. In the initialization, a basic structure of six free bases is created as a starting point. To this structure, a number of helices are added. Each of these helices can be considered a separate domain, since the final structure will have no interaction between the nested helices on these domains. The same effect could be achieved by creating separate structures and concatenating them.

Legend:

NewStructure: The structure being generated;
HelicesEnds: A structure which records positions where **()** is added;
NumberDomains: The number of helices added to the initialized *NewStructure*;
MultiLoopLocations: A structure which records positions where multiloops have been added;
m, n, x: Parameters set or randomly generated to determine the length of each structure.

Functions:

Initialize: Initializes *NewSequence* to **......** and an appropriate number of helices are added, depending on the number of domains;
AddHelices: Adds $(_{\mathbf{m}})_{\mathbf{m}}$ to the structure at a specified position, recording where multiloops or hairpins are to be added;
AddMultiloops: Adds $._{\mathbf{n}}$ to the structure at a specified position, recording where helices are to be added;
AddHairpin: Adds $._{\mathbf{x}}$ to the structure at a specified position, between **()**;
AddFreeBaseAfter: Adds **.** at a specified position.

Proposed changes

The mechanism by which free bases are added to the structure does not reflect any significant biological intuition. A number of much more elegant functions for the generation of bulges (free base inserts of length greater than one) or loops (free base inserts on both sides of a helix at complementary positions) could be generated with little effort.

```
procedure GenerateStructure
    input: parameters for length of elements added
    output: NewStructure
    (NewStructure, MultiloopLocations) = Initialize (NewStructure, NumberDomains);
    (NewStructure, PositionOfMultiloop) = AddMultiloops (NewStructure, NumberMultiloops);
    (NewStructure, HelicesEnds) = AddHelices (NewStructure, HelicesPerMultiLoop);
    while (more multiloops desired)
        MultiloopLocations = AddMultiloops (NewStructure, HelicesEnds);
        foreach (Helices desired per Multiloop)
            HelicesEnds = AddHelices (NewStructure, MultiLoopLocations);
        end foreach;
    end while;
    while (HelicesEnds still contains positions)
        NewStructure = AddHairpin (NewStructure, HelicesEnd);
    end while;
    foreach (PositionInStructure)
        if (Rand%101 < AcceptProbability)
            NewStructure = AddFreeBaseAfter (BaseInStructure);
        end if;
    end foreach;
    return NewStructure;
end GenerateStructure.
```

Figure 11: Pseudocode for the Random Structure Generator

6 Experiments and Results

In this section, we study the performance of IRNAF. We investigate its sensitivity to the noise parameters p_a and p_c, and determine the importance of hierarchical splitting and heuristic initialization. Finally, we demonstrate superior performance over the Vienna inverse folding algorithm [16].

6.1 Conventions and Data for Empirical Analysis

We used two data sets for the empirical analysis. The first data set, `rand_gen`, is created with the random structure generator introduced in Section 5.4. The structures vary in number of domains and length. None of them has any bulges. The second data set, which we refer to as `rrna` is created from rRNA sequences obtained from the Ribosomal Database Project [23]. We folded sequences obtained from there using the *fold* function from the Vienna RNA Secondary Structure Package [25], which implements Zuker's algorithm [15] to fold a sequence into the minimum energy structure. This forward folding algorithm is the one used in our algorithm to evaluate sequences; thus, by obtaining structures as folds of sequences, we can be certain that at least one solution to each of the structures exist. Shorter structures are extracted from these structures by means of the hierarchical splitting described in Section 5.2. We refer to complete rRNA structures by the "definition" used in the original database [23].

The algorithms we investigate are highly randomized; thus, it is advantageous to base their analysis on more than one try. Runtime distributions (RTDs) [30] are a good way to visualize the behaviour of randomized algorithms. Unfortunately, the execution of the algorithms we are comparing is often so slow that we cannot afford to run them too often. Thus, we mainly base our analysis on the execution of 10 runs per algorithms and instance. An exception are the 2D sensitivity surface plots in Figure 12; to reduce the noise in the plots we ran the algorithm for 100 tries for each parameter setting.

Each try of an algorithm is cut off after 100 seconds if not otherwise stated. If the run finishes earlier and solves the problem, it terminates; if it does not solve the problem but is still within the time limit, it is simply restarted. When the cutoff time is reached and the current run finishes, the best result obtained so far is returned together with the total runtime.

For structures longer than 300 bases, the Vienna inverse algorithm rarely finds a solution. Thus, in order to do a proper analysis, we need to represent both runtime and distance. A very computationally inexpensive way to achieve this is chosen here. For various instances, we only do a single try (which still can entail more than one run if it stays within the time limit); however, for structures longer than around 700 bases, the Vienna inverse algorithm still often does not terminate after hours of CPU time. In these cases, we terminated the runs manually and report the user time after which the runs were terminated.[2] The shortcomings of only performing one try are obvious; however, given the restricted amount of time and the apparent computational hardness of inverse RNA folding, this is the best compromise we could achieve.

6.2 Characteristics of IRNAF

In this section, we describe experiments we carried out in order to set the noise parameters of IRNAF and determine which components of the algorithm are important for its performance.

Our first experiment was to study IRNAF's noise sensitivity. Figure 12 shows surface plots for various noise settings and problem instances. Only one instance from **rand_gen** was tested due to the longer runtime of instances from this data set.[3] Due to the small number of tested structures only limited conclusions can be drawn. Values of $p_a > 0.3$ seem to result in bad performance; also, too high values of p_c seem to perform worse than reasonably small ones. We do not want to draw premature conclusions regarding the optimality of any values, but fix the values to $p_a = 0.2$ and $p_c = 0.2$ for the rest of the paper; these values seemed to be reasonably robust.

[2]All experiments reported here were carried out on Pentium III double processor machines with 1 GHz; each terminated run constantly had a CPU load on one processor of over 95%.

[3]We suppose the longer runtimes are solely based on the longer length of the shortest sequences in `rand_gen`. Unfortunately, we did not generate structures shorter than 200 bases for this data set. For comparison, the structures `rrna100*` have lengths between 20 and 100.

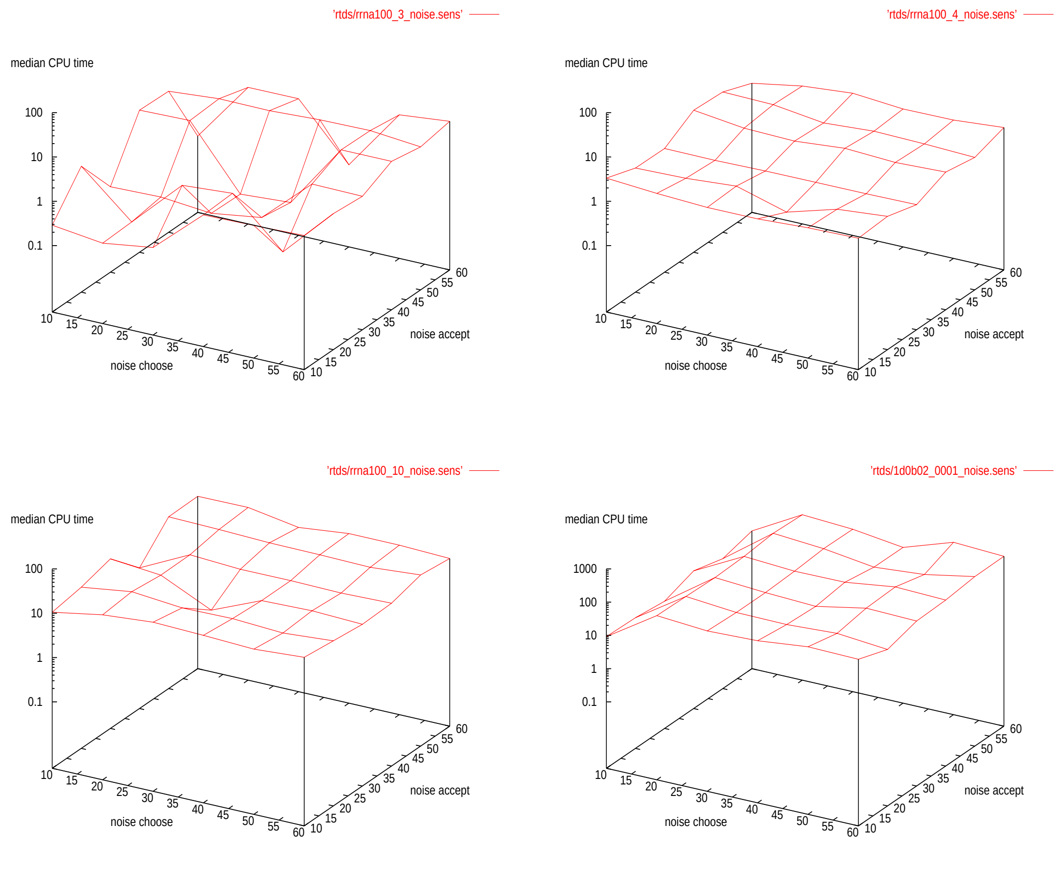

Figure 12: Noise sensitivity for three secondary structures from the **rrna** data set and one structure from the **rand_gen** data set (bottom, right); noise accept $= 100 * p_a$, noise choose $= 100 * p_c$

As can be seen in figure 13, the RTDs of IRNAF do not always look like typical RTDs of SLS algorithms. In particular, some of them do not have a clear exponential shape; however, they come close to mixtures of exponentials. The RTDs of the Vienna inverse folding algorithm look more regular, which suggests less stagnation behaviour. Note that the algorithms are restarted when they end unsuccessfully. Stagnation behaviour of the SLS algorithm could explain the curves: in the beginning of each run, it is more likely to find a solution then in a later stage. This would hinge on decreasing the number of flips that each try is allowed to perform before cutoff. However, when dealing with large structures, the initialization is done for the whole sequence, such that a complete restart with a new initialization can not be performed too often. Further analysis needs to clarify this issue; since numerous restarts are infeasible for longer structures, stagnation behaviour would have disastrous effects on the algorithm. An improvement on this would thus appear especially promising.

The structures for which RTDs are plotted are:

```
((..((...)).)) (weird)

((((..((.(.((((....)))).).))...)))) (rrna100_11)

((((..((.((((((....)))))).))...)))) (rrna100_13)

(((((..(((((..(.(.(((......)))....).)..)))))..))))) (rrna100_4)

((((........(((........))).....((..((((....))))..))....)))) (rrna100_16)

((((((((((((((..((((.....))))..))))))...((....))...)))))))) (rrna100_3)

(((((((.(((((((..(((.......)))..)))))))...((....))....))))))) (rrna100_6)

((...((((((..(((((.(((..((((.((((....))))))))..)))......(((....))))))))...)))))).)) (rrna100_9)
```

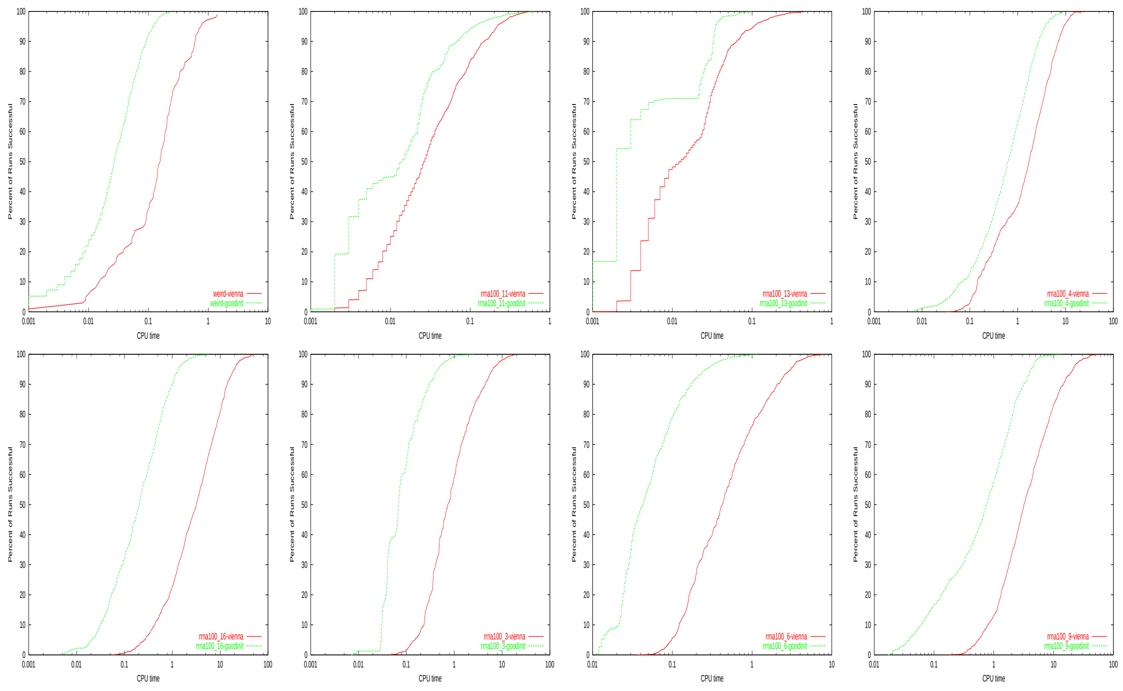

Figure 13: Empirical run-time distributions (RTDs) for IRNAF and the Vienna inverse folding algorithm. Problems from the domain `rrna`.

In Figure 14, we evaluate how the performance of IRNAF degrades when omitting the splitting of large structures(left) and the heuristic initialization(right). We come to the intuitive conclusion that the splitting becomes more important when the structures are getting longer (15 of the 20 longest structures were not finished within the cutoff time despite the fact that these structures were smaller than 300 bases). The heuristic initialization on the other hand seems to yield improvements rather independently from the structure length.

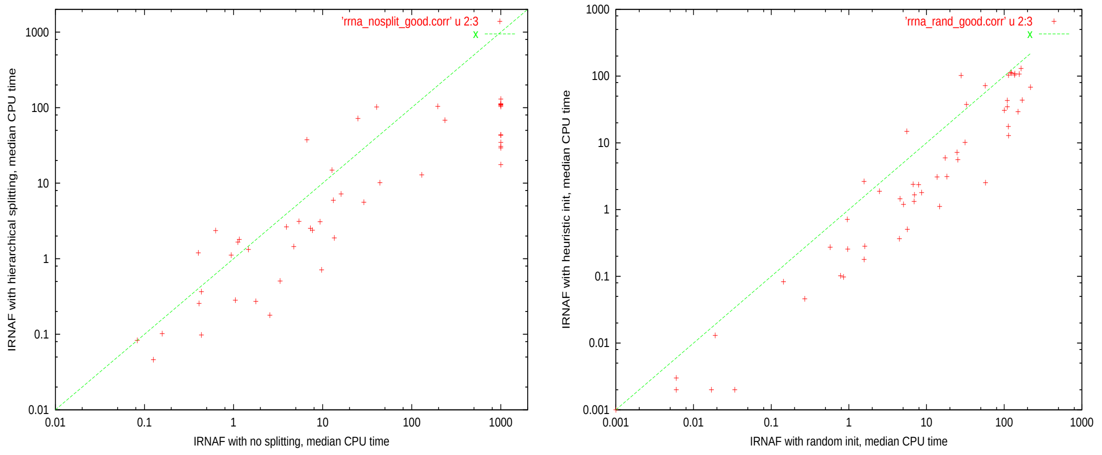

Figure 14: Improvements due to splitting (left) and heuristic initialization (right). For the splitting, note that the 15 points (from **rrna300**) with x-value 1000 did not succeed after a total runtime of 10000 CPU seconds for 10 runs. Problem domain: **rrna**

6.3 IRNAF vs Vienna inverse folding

Figure 15 compares IRNAF with the Vienna inverse folding algorithm. For the short structures in this plot, we observe a performance improvement of about an order of magnitude. For the data set **gen_rand**, it can be seen that the performance difference grows with structure hardness. This reflects the the better scaling behaviour of IRNAF and we believe data set **rrna** would show the same trend for harder instances.

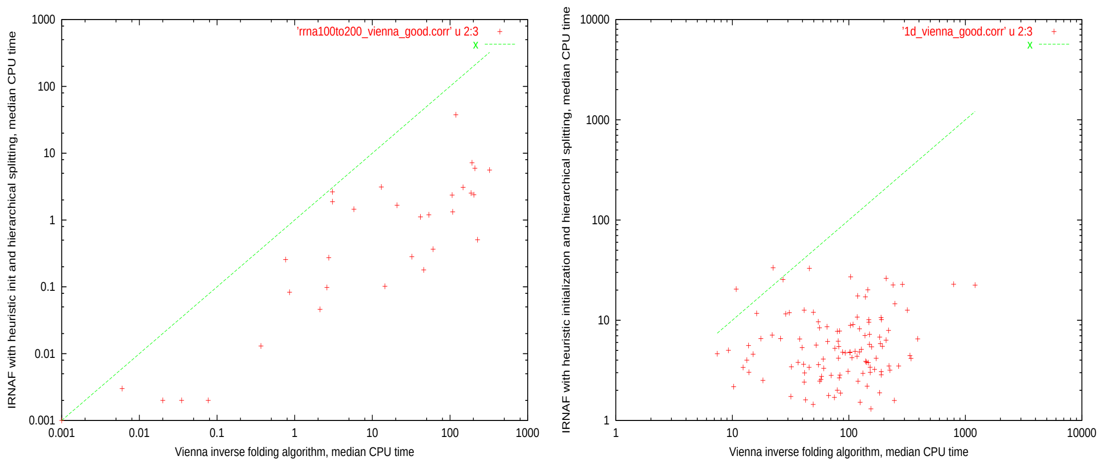

Figure 15: Comparison of IRNAF and the Vienna algorithm. Problem domains: **rrna** (left), **rand_gen** (right). For the left plot, note that the cutoff time per run was set to 100 CPU seconds, meaning all runs corresponding to a data point with an x-value > 100 are not guaranteed to have succeeded with Vienna

In Tables 1 and 2, we give the results of only one run of IRNAF, as compared to one run of the Vienna inverse folding algorithm for some complete rRNA secondary structures. We know about the limits of this approach when dealing with randomized algorithms, but do hope to give first evidence on the superior behaviour of IRNAF. A more detailed analysis needs to be done in the near future. Entries marked with an @ correspond to incomplete runs of Vienna that had to be interrupted manually. For each of these instances, the user time until interruption is given. Entries marked with a + correspond to a run of our algorithm that returned an error message. We believe this quirky bug is most likely due to the unstable C command *strcpy*; this issue will be dealt with as soon as possible.

7 Possible Improvements and Future Work

Various improvements to this algorithm may be possible. First, a possible improvement in running time, one could use a custom implementation of the forward algorithm in which the matrix V(i,j), containing the dynamic programming solutions to the energy minimization, would be updated rather than completely re-calculated. The recalculation of V(i,j) is wasteful in large parts when many of the matrix elements are unchanged between calls from either the heuristic or the SLS iterations. Note, however, that the complexity of such an update would still be $O(n)$, where n is the number of bases in the structure.

The second possible improvement would be to use a dynamic value for the constant k in the initialization phase, rather than a single fixed value. Thus, the number of possible combinations available could be expanded for longer helices, which may also allow for larger structures with more helices to be solved.

With respect to the SLS, we believe there is room for improvement. The optimal noise parameter values have only been estimated from 4 short structures. It is likely that the optimal values for a long structure with a lot of recursive calls might have quite different optimal settings. Furthermore, the basic SLS might be improved, for example using a TABU-like mechanism as introduced for Constraint Satisfaction problems in [28].

Future work on this algorithm should include a more detailed RTD-based empirical analysis. Also, the testing done to date has mainly been performed on ribosomal sequences which have been converted to structures using a forward folding function from the Vienna Package. However, a wider variety of structures should be used and evaluated. What is clearly needed is a standard data set of structures for comparative empirical analysis. To this end, a sequence generator, which is able to create randomly generated structures, was produced. Interestingly, some structures that are produced by the random structure generator cannot be solved by IRNAF (nor the Vienna algorithm). It might be interesting to investigate if these structures are just particularly hard or simply not viable. A complete algorithm could be used to answer these questions.

Furthermore, the algorithmic ideas of this approach might carry over to more advanced problems like the design of multistable RNA molecules, as dealt with in [19]. Stochastic Local Search methods are very likely to find application in this domain as well.

Definition of structure	Length	IRNAF distance	IRNAF time	Vienna Inverse distance	Vienna Inverse time
Sargasso Sea 10m depth bacterioplankton DNA clone BDA1-10	299	0	786.080	9	1390.773
Clone pM9-23	299	0	22.258	12	733.410
Clone pB7-128R	299	0	378.598	4	1555.210
str. DCM FREE	299	0	170.783	2	698.762
Unidentified soil bacterium from soybean field clone FIE3	299	0	5.023	2	867.784
Leptospira interrogans str. 94-7997013	299	4	1032.644	15	1393.612
Str. NR64, bacterium expressing pSym plasmid	300	0	7.989	2	1272.369
Acetobacter LMG 1529 LMG 1529	348	0	16.608	37	4039.679
Ochrobactrum BL200-8 str. BL200-8	357	0	7.781	2	262.201
Str. Marine sediment sample A2 from beneath salmon fish cage, first2	357	0	245.820	4	1747.893
Methanobrevibacter FMBK1	359	0	715.433	@	3:02h @
Micrococcus uncultured bacterium SY2-74	359	0	15.074	22	2489.510
Anabaena uncultured bacterium SY2-21	359	0	206.738	10	1548.105
Octopus Spring microbial mat DNA from Yellowstone NP clone Type E	417	8	1095.179	10	3153.804
Prevotella albensis str. M384 DSM 11370 (T)	419	28	1096.816	17	2678.361
Unnamed organism	419	0	16.884	20	2435.991
Clone 3-25	419	0	244.009	8	2805.905
Mud Volcano area of Yellowstone NP ("Black Pool") hot spring DNA	474	0	140.628	2	4144.913
Unnamed organism	478	31	1071.219	20	10078.846

Table 1: Runtimes and distances for IRNAF and the Vienna inverse folding algorithm for various rRNA secondary structures

Definition of structure	Length	IRNAF distance	IRNAF time	Vienna Inverse distance	Vienna Inverse time
Str. ST9630	479	2	1146.429	11	4093.585
Unnamed organism	479	0	889.529	37	8792.324
Clone CRO-FL22	479	0	640.432	20	2586.165
Clone CRE-FL72	479	34	1066.446	27	7226.848
Clone LBS8	479	5	1091.351	8	2855.989
Unnamed organism	479	0	715.576	14	2947.360
Neisseria lactamica DSM 4691	479	3	1005.220	41	4917.567
Stenotrophomonas sp. str. P-9-8	534	+	+	20	9790.491
Clone vadinIA59	539	0	150.109	8	8820.554
Nitrobacter Nb4 str. Nb4	659	21	1098.328	15	7005.243
Wolbachia pipientis	659	+	+	8	4750.398
Clone ST1-4	776	0	100.409	37	9819.491
Bradyrhizobium sp. str. 283A	780	4	1063.200	@	4:02h @
Sulfolobus acidocaldarius str. N8	899	67	2901.292	@	3:48h @
Spirochaeta sp	899	+	+	@	4:12h @
Str. 1200m deep water sample of Lake Baikal, Russia	960	+	+	@	6:36h @
Dendrodoa grossularia	1019	2	2074.461	16	19847.821

Table 2: Runtimes and distances for IRNAF and the Vienna inverse folding algorithm for various rRNA secondary structures

8 Conclusions

We have introduced a new algorithm for the inverse RNA folding problem, which we call IRNAF. IRNAF demonstrates a significant increase in both the accuracy and speed as compared to the Vienna inverse folding algorithm. The use of both an SLS algorithm with a heuristic initialization provides a good starting point in the exploration of alternate means of solving a difficult biological problem. It validates both the use of *a priori* knowledge from a biological perspective as well as the application of modern computational methods to a biologically significant problem.

We have demonstrated that the simple biological insights which have been incorporated into the heuristic algorithm are a good starting point for further work. Although they do not effectively solve this problem, they are able to generate a good initial assignment for the SLS algorithm. Through key insights, we were able to improve the heuristic algorithm and demonstrate that simple biological principles can have a significant optimizing impact upon the generation of solutions to this problem.

We have shown here that for a great number of cases, the SLS optimization is able to propose changes which yield optimal solutions, in which the distance between the input structure and the structure generated by folding the solution sequence is zero. Although not all structures could be solved by this method, in particular, artificially generated hard sequences, the increases in performance, the size of the structures that can be solved and the number of sequences which can be solved demonstrate the overall validity of this approach. It is hoped that this algorithm can provide a significant contribution to a biologically difficult problem which may have great relevance in the future.

References

[1] George Gamov, *Possible relation between deoxyribonucleic acid and protein structures* Nature 173: (1954), p. 318.

[2] George Gamov, *Information transfer in the living cell* Scientific American 193:4 (1955), pp. 70-78.

[3] Palade, G.E. (1955) *A small particulate component of the cytoplasm* J. Biophys. Biochem. Cytol. 1:59-68

[4] P J Perry, T C Jenkins (1999) *Recent Advances in the Development of Telomerase Inhibitors for the Treatment of Cancer* Exp. Opin. Invest. Drugs, 8 (12), 1981-2008.

[5] Berendsen H.J. (1998) *A glimpse of the Holy Grail?* Science 282(5389):740-4

[6] Lyngso, R.B., Pederson, C.N., 2000 *RNA pseudoknot prediction in energy-based models* J. Comput. Biol. 7(3-4):409-27

[7] Cech, T.R., A.J. Zaug, and P.J. Grabowski (1981) *In vitro splicing of the ribosomal RNA precursor of Tetrahymena: involvement of a guanosine nucleotide in the excision of the intervening sequence* Cell 27:487-496

[8] Breaker, R. R.; Joyce, G. F. Trends in Biotechnol. 1994, 12, 268-275. *Inventing and improving ribozyme function: Rational design versus iterative selection methods.*

[9] Cech, T. R. Science 1987, 236, 1532-1539. *The chemistry of self-splicing RNA and RNA enzymes.*

[10] Campbell TB, Sullenger, BA. *Alternative approaches for the application of ribozymes as gene therapies for retroviral infections.* Advances in Pharmacology 1995;33:143-178.

[11] Christoffersen, R. E.; Marr, J. J. J. Med. Chem. 1995, 38, 2023-2037. *Ribozymes as human therapeutics.*

[12] Usman, N.; McSwiggen, J. A. Annu. Rep. Med. Chem. 1995, 30, 285-294. *Catalytic RNA (ribozymes) as drugs.*

[13] Cech, T. R. Curr. Opin. Struct. Biol. 1992, 2, 605-609. *Ribozyme engineering.*

[14] Schuster P., (1995), How to search for RNA structure. *Theoretical concepts in evolutionary biotechnology.* J. Biotechnol. 41(2-3):239-57

[15] Run B. Lyngso, Michael Zuker and Christian N. S. Pedersen, 1999. *An Improved Algorithm for RNA Secondary Structure Prediction*, BRICS Report Series, RS-99-15, Department of Computer Science, University of Aarhus.

[16] Ivo L. Hofacker, Walter Fontana, Peter F. Stadler, L. Sebastian Bonhoeffer, Manfred Tacker and Peter Schuster, 1994. *Fast folding and comparison of RNA secondary structures*, Chemical Monthly, 125: 167–188.

[17] Martin Tompa, *Lecture Notes on Biological Sequence Analysis*, Deparment of Computer Science and Engineering, University of Washington, `http://www.cs.washington.edu/education/courses/527/00wi/`.

[18] Michael Zuker's web site on RNA secondary structure prediction. `http://bioinfo.math.rpi.edu/~zukerm/rna`

[19] Christoph Flamm, Ivo L. Hofacker, Sebastian Maurer-Stroh, Peter F. Stadler and Martin Ziehl, 2001. *Design of multistable RNA molecules*, RNA(2001), Cambridge University Press, 7:254–265.

[20] Elena Rivas and Sean R. Eddy, 1999. *A Dynamic Programming Algorithm for RNA Structure Prediction Including Pseudoknots*, Journal of Molecular Biology 1999, 285, 2053–2068.

[21] Holger H. Hoos, *Stochastic Local Search - Methods, Models, Applications*, PhD thesis, Department of Computer Science, Darmstadt University of Technology, 1998.

[22] Holger H. Hoos and Thomas Stützle, *Stochastic Local Search - Foundations and Applications*, Morgan Kaufmann, to appear in 2003.

[23] The Ribosomal Database Project, `http://rdp.cme.msu.edu/download/SSU_rRNA/unaligned/SSU_Unal.gb`

[24] A. M. Zucker, B. D. H. Mathews and C. D. H. Turner, *Algorithms and Thermodynamics for RNA Secondary Structure Prediction: A Practical Guide*, RNA Biochem & Biotech, J. Barszewski &B.F.C. Clark, eds, Nato ASI Series , Kluwer Academic Publishers (1999)

[25] The Vienna RNA Secondary Structure Package.
`http://www.tbi.univie.ac.at/ ivo/RNA/`

[26] Bart Selman and Henry A. Kautz and Bram Cohen. Noise Strategies for Improving Local Search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pages 337–343, AAAI Press, 1994.

[27] David A. McAllester and Bart Selman and Henry A. Kautz. Evidence for Invariants in Local Search. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 321–326, AAAI Press, 1997.

[28] Olaf Steinmann, Thomas Stützle and Antje Strohmaier. Tabu Search vs. Random Walk In *KI-97:Advances in Artificial Intelligence*, Springer Verlag, LNCS, Vol. 1303.

[29] Holger H. Hoos. On the Run-time Behaviour of Stochastic Local Search Algorithms for SAT. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 661–666. AAAI Press, 1999.

[30] Holger H. Hoos and Thomas Stützle. Local Search Algorithms for SAT: An Empirical Evaluation. In *Journal of Automated Reasoning*, Volume 24, Number 4, pages 421–481, 2000.