# Partial-Order Alignment of RNA Structures

by Ethan Bromberg-Martin

as an undergraduate thesis in Computational Biology

## Outline

## 0. Abstract

Multiple alignment of DNA, RNA, and protein sequences is an invaluable tool for comparative genetics. Some RNAs have functions that depend on structural features as well as their sequences, so several methods have been proposed for aligning multiple RNA secondary structures. One trait they share is that their alignment positions have a total ordering. (Lee and Grasso, 2002) have shown that biological sequence alignments are more naturally represented under a partial ordering. Partial-order graphs represent insertions and deletions without the pitfalls of 'gapped' alignments. I argue that a partial-order formulation has even greater benefits for RNAs; RNA structural alignment is fundamentally partially ordered. Total-order representations are prone to misalignments because they represent “sequence *and* structure”, not “sequence *or* structure”.

In this thesis, I introduce RNA partial-order alignments (RNAPOAs), a new type of multiple structural alignment. I show how to construct and align RNAPOAs based on two popular representations for RNA structures: arc-annotated strings and ordered forests. RNAPOAs faithfully represent both shared similarities and individual variations of aligned sequences and structures. This allows RNAPOAs to support and improve a variety of advanced algorithms, including optimal alignment of base-pairing probability matrices and approximate alignment of tertiary structures. Partial-order algorithms are as efficient as their total-order counterparts, so these advantages do not come at a cost in algorithmic complexity.

# 1. Overview

## 1.1 Motivation

A *multiple sequence alignment* is a representation of the similar features of a set of DNA, RNA, or protein sequences, and can be used to infer their shared functions and evolutionary history. Sequence alignments are crucial tools for biological research, and so there has been a great deal of effort devoted to making them faster and more accurate. Yet the functions of RNAs and proteins depend on their 3-dimensional structures as well as their sequences, so it is worthwhile to consider comparison methods that include both sequence and structure information.

There are a wide variety of algorithms for comparing RNA structures, often restricted to *RNA secondary structure* because higher-level structure tends to be computationally intractable. A flurry of methods have recently been proposed for aligning multiple RNA secondary structures (Wang and Zhang, 2004; Hochsmann *et al.*, 2004; Hofacker *et al.*, 2004; Liu *et al.*, 2005). One assumption they share with most sequence alignment approaches is that the positions in an alignment should have a total ordering. Specifically, each position in each aligned structure must be matched with some positon in every other structure. If a particular feature only exists in some of the structures, artificial 'gap' entities have to be inserted into every other structure to preserve the total ordering. This introduces a degeneracy: we can choose many different ways to put gaps into an alignment without changing which features are aligned with each other. However, our choice of gaps will have a large effect on how we build the alignment, and hence on the quality of the finished product.

As recent work by Lee and Grasso (Lee *et al.*, 2002; Grasso and Lee, 2004) demonstrates, alignments of sequences are more naturally represented by *partial-order multiple sequence alignments* (PO-MSAs). A PO-MSA is a directed acyclic graph that contains each aligned sequence as a subgraph. Rather than using 'gaps', mutated regions are simply alternate paths through the graph. Hence a PO-MSA reflects the fact that these mutated regions have no true ordering with respect to each other.[1] This property gives rise to several other advantages; PO-MSAs can be computed very quickly, and whenever a new sequence is added to a PO-MSA, we can consider its optimal alignment with each other sequence.

Since an RNA structure alignment essentially includes a sequence alignment, it stands to reason that partial-order formulation should be at least as beneficial as it is for sequences. In addition, alignments of certain structural features make sense as partial orderings but not as total orderings. For example, a few mutated nucleotides can push a region of an RNA into a different structural state. Even if we do not know how both sequence and structure contribute to an RNA's functions, a total order alignment forces us to prioritize the two: if we match the structure then we will have a poor match for the sequence, and vice versa. What we really want our alignment to say in this case is "they

1 Well, they may have an ordering, but not one that can be deduced from the sequences in the alignment.

have this structural variant *or* that structural variant, but similar sequences". This type of pattern-matching is captured by alternate paths through a partial-order graph.

## 1.2 Main Results

I introduce RNA partial-order alignments (RNAPOAs), a representation for multiple alignments of RNA structures based on partial-order graphs. RNAPOAs have the same advantages over current RNA alignments that partial-order sequence alignments have over traditional, totally-ordered representations. The partial-order formulation also has several additional benefits in the domain of RNA secondary structure: RNAPOAs can represent alignments of certain structural features that total-order representations cannot. In fact, I argue that RNA structure alignments are inherently partially ordered.

I then develop algorithms for optimal global and local alignment of two RNAPOAs. RNAPOAs come in two flavors. String-RNAPOAs are based on the arc-annotated strings of (Bafna *et al.*, 1996), and are aligned with algorithms in the vein of (Wang and Zhang, 2001; Hofacker *et al.*, 2004). Forest-RNAPOAs are based on the ordered forests of (Hochsmann *et al.*, 2003), and are aligned with algorithms like those of (Jiang *et al.*, 1995; Jansson *et al.*, 2004). If any of these partial-order algorithms is used for unstructured RNAs, it reduces to the partial-order sequence alignments of (Grasso and Lee, 2004). The time/space complexity of these algorithms is essentially the same as for total-order alignment. The complexity increases for RNAPOAs that contain dissimilar RNAs, for the same reason that total-order alignments become slower if they contain many 'gaps'. In most real-world cases this increase should be bounded by a small constant factor. In addition, I also describe algorithms for approximate alignment of RNAPOAs that have tertiary structure, based on a similar method of (Wang and Zhang, 2001). This method also allows us to align 'alternate paths' within RNAPOAs, addressing a subtle source of error in current methods for partial-order alignment. In addition, I discuss several strategies for constructing multiple alignments from a series of pairwise alignments.

Finally, I demonstrate RNAPOA by aligning a set of related RNA secondary structures. It is difficult to objectively evaluate the quality of partial-order alignments on 'ground truth' data, since one partial-order alignment represents a very large number of different total-order alignments. This is especially true for RNA secondary structures, for which high-quality phylogenetic alignments are typically expressed with a single, total-order consensus sequence. However, these preliminary results show how RNAPOAs can represent alignments with structural variations that cannot be captured by total order alignments.

In summary, partial-order graphs are natural representations for RNA secondary structure alignments which overcome the limitations of traditional methods. This improvement in representation yields an improvement in alignment accuracy, and in most cases does not require an increase in complexity.

### *1.3 Organization of this thesis*

The first section of this thesis is this introduction.

The second section presents background information about sequence and RNA structure alignment. I begin with a brief primer on sequence alignment and the traditional 'row-column', progressive profile formulation of multiple alignment (Higgins and Sharp, 1988). This leads into a dicussion of the advantages of the recent partial-order formulation of Lee and Grasso. I will then explain the basics of RNA structure and motivate the problem of RNA structure alignment. I will discuss the existing methods for pairwise and multiple alignment of RNA structures, with detailed attention to the algorithms of (Wang and Zhang, 2001; Jiang *et al.*, 1995; Jansson *et al.*, 2004) which I will build on in this paper. Finally, I will give some thoughts on current methods for RNA structural alignment, and how they can be improved with a partial-order formulation.

The third section introduces a graph representation of RNA partial-order alignments, or RNAPOAs for short. RNAPOAs are generalizations of partial order sequence alignments and RNA secondary structure alignments. Each path through an RNAPOA is a secondary structure, and each aligned RNA exists as a path in the RNAPOA. They come in two flavors: one based on arc-annotated strings, and one based on ordered forests. I will then develop algorithms for global and local pairwise alignment of RNAPOAs, including variants that handle different scoring schemes and advanced types of structural information. I will also present an approximation algorithm for aligning 'alternate paths' in RNAPOAs (e.g. tertiary structure). I will conclude with a brief discussion of various approaches for building multiple alignments by repeated application of the pairwise algorithms.

The fourth section presents the experimental results of the POSSA algorithm on sets of related RNA secondary structures.

The fifth section is a discussion of the merits and drawbacks of the RNAPOA representation and alignment algorithms. I will also talk about some interesting extensions to the RNAPOA algorithm that ought to be the subject of future research.

The sixth section has references for all of the scholarly work that I've cited.

# 2. Background

This section presents background information about sequence and RNA structure alignment. I begin with a brief primer on sequence alignment and the traditional 'row-column', progressive profile formulation of multiple alignment. This leads into a dicussion of the advantages of the recent partial-order formulation of Lee and Grasso. I will then explain the basics of RNA structure and motivate the problem of RNA structure comparison. Finally, I will discuss existing methods for aligning multiple RNA structures, and explain how these methods can be improved by using a partial ordering.

## *2.1 Alignment of two sequences*

Alignment of DNA, RNA, and Protein sequences is a common operation in sequence analysis. An 'alignment' is just a correspondence between similar regions of a set of nucleotide sequences. If several sequences are globally similar, then they are likely to share a common ancestor and to have similar biological functions. Let us consider two two RNA sequences whose nucleotides are represented by letters in {a,u,g,c} :

```
S = augcgacagu
T = cagagag
```

We can see that a region of *T* (“gagag”) is very similar to a region of *S* (“gacag”). One way to associate these regions is to insert *gap* characters ('-') into the two sequences until they are matched with each other, like so:

```
S = augcgacagu
T = --cagagag-
```

Now we have a traditional *row-column* alignment. A row-column alignment is a matrix where each row represents a sequence and each column represents a set of matched nucleotides. Each row of the alignment has to have all of its sequence's nucleotides in the correct order, possibly with gaps mixed in. Each column that doesn't have a gap represents a mapping between a nucleotide in the top sequence and a nucleotide in the bottom sequence. I will typically refer to sequences and alignments in terms of 'positions'; the *i*-th base in a sequence is its *i*-th position and the *j*-th column of an alignment is its *j*-th position.

The key to an RC alignment is its gaps, since gaps are what force similar regions into register. If the gaps are inserted properly then similar regions are matched with each other and the alignment is an informative mapping between the two sequences. If gaps are inserted poorly then the alignment tells us nothing. There are many ways of measuring the goodness of an alignment; here, I will explain the most well-known method.

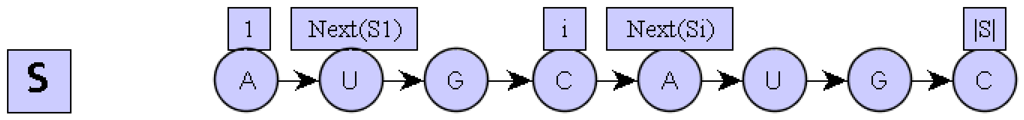
Let us consider each pair of sequence positions ($p_S$, $p_T$), respectively from sequences $S$ and $T$, that are placed in the same column of the alignment. If $p_S = p_T$, then we say they *match*. If $p_S \neq p_T$, then we say they *mismatch*. This is also called a substitution or replacement, since it corresponds to a sequence suffering a point mutation that replaces one nucleotide with another. If $p_S$ is a '-' gap character, then we say $p_T$ has been *gapped*. A stretch of several gaps in a row is an *indel*, short for insertion/deletion, which represents mutations that add or remove several nucleotides at once. Every column has to have at least one nucleotide from one of the sequences; we prohibit columns from having only gaps. The alignment above has 4 matches, 3 mismatches, and 2 indels with 3 total gaps.

Now we can define the problem of finding a good alignment of $S$ and $T$ as an optimization over all possible ways to insert gaps into them. We'll define a scoring function for an alignment that gives bonuses for matches and penalties for mismatches and indels. We then try to find the alignment which has the best score. The simplest scoring model of this type is as follows. We define a scoring function $score(p_S,p_T)$ for matching position $p_S$ in sequence $S$ with position $p_T$ in sequence $T$. The score of an alignment is just the sum of these scores over all positions in the alignment, i.e. $\sum_{p_S \in S,\, p_T \in T} score(p_S, p_T)$ . To encourage good alignments, we must define *score* to reward matches (when p = p') and penalize mismatches and gaps.

Under this scoring model, it is well known that we can use dynamic programming to find the optimal alignment in quadratic time and space (Needleman and Wunsch, 1970). Here I will present this algorithm in terms of aligning two graphs; this will allow us to easily generalize this algorithm for more complicated graph alignment problems later on in this thesis. We need some definitions:

$S$ is a sequence of $m$ nodes, $(s_1,s_2,...,s_m)$
$T$ is a sequence of $n$ nodes, $(t_1,t_1,...,t_n)$
$next(s_i) = s_{i+1}$, the next node in S after $s_i$. $next(s_m)$ = empty.
$(s_i,s_j)$ denotes a substring of $S$: a pair of nodes in $S$ such that $i < j$.
$(s_i,s_m)$ is a suffix of $S$

We'll find the optimal alignment by considering optimal alignment of each suffix ( $s_i,s_m$) $S$ with each suffix $(t_k,t_n)$ of $T$. We can treat the sequence $S$ = “augcaugc” as a graph where each position $s_i$ has an edge $next(s_i)$ to the next nucleotide:



Let us define $ALIGN((s_i,s_m),(t_k,t_n))$ to be the optimal score for aligning the two suffixes $(s_i,s_m)$ in $S$ and $(t_k,t_n)$ in $T$. If we can compute $ALIGN((s_1,s_m),(t_1,t_n))$, then we'll have the optimal score for aligning the whole of $S$ with the whole of $T$. In order to compute $ALIGN((s_i,s_m),(t_k,t_n))$, we'll break it into three subalignment; three alignments of smaller pairs of suffixes. These correspond to three mutation operations that we could use to transform S into T: *replacement*, *insertion*, and *deletion*.

For the first case, what if $ALIGN((s_i,s_m),(t_k,t_n))$ puts $s_i$ and $t_k$ in the same column? Then the bases are matched or mismatched with a score of $score(s_i,t_k)$. I will call this a replacement, since to transform *S* into *T*, we would have to *replace* $s_i$ with $t_k$.[2] What about the rest of the columns of the alignment? They must be the optimal alignment of the rest of $(s_i,s_m)$ with the rest of $(t_k,t_n)$, i.e. $ALIGN((next(s_i),s_m),(next(t_k),t_n))$. So the optimal score of an alignment for a replacement is $score(s_i,t_j)$ for matching or mismatching the bases plus $ALIGN((next(s_i),s_m),(next(t_k),t_n))$ for the rest of the alignment.

For the second case, what if $ALIGN((s_i,s_m),(t_k,t_n))$ doesn't put $t_k$ in the same column as any nucleotide in S? Then $t_k$ must be gapped at a cost of $score(-,t_k)$. We say that its position represents an insertion of $t_k$, since to transform S into T, we would have to *insert* the nucleotide $t_k$. What about the rest of the alignment? We have to align $(s_i,s_m)$ with the remainder of $(t_k,t_n)$: $ALIGN((s_i,s_m),(next(t_k),t_n))$

For the third case, what if $ALIGN((s_i,s_m),(t_k,t_n))$ doesn't put $s_i$ in the same column as any nucleotide in *T*? Then this is just like an insertion, but we put a gap opposite to $s_i$. We call this a deletion; to transform *S* into *T* we would have we to *delete* $s_i$ from *S*.

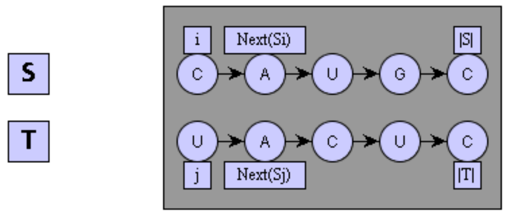
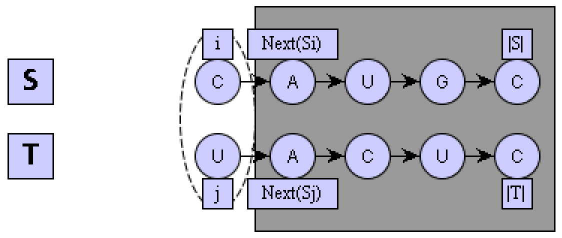
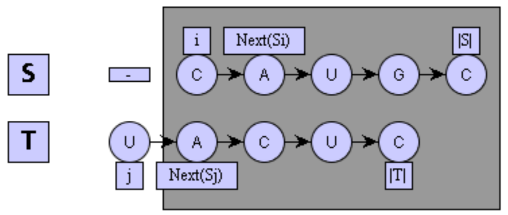
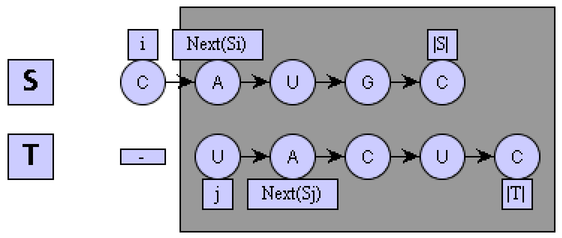
| | |
|---|---|
|   S<br>T<br>i<br>Next(Si)<br>\|S\|<br>C A U G C<br>U A C U C<br>j<br>Next(Sj)<br>\|T\| Subalignment of two suffixes $ALIGN((s_i,s_m),(t_k,t_n))$ |   S<br>T<br>i<br>Next(Si)<br>\|S\|<br>C A U G C<br>U A C U C<br>j<br>Next(Sj)<br>\|T\| Case 1: replace $score(s_i,t_k)+ALIGN((next(s_i),s_m),(next(t_k),t_n))$ |
|   S<br>T<br>-<br>i<br>Next(Si)<br>\|S\|<br>C A U G C<br>U A C U C<br>j<br>Next(Sj)<br>\|T\| Case 2: insert: $score(-,t_k)+ALIGN((s_i,s_m),(next(t_k),t_n))$ |   S<br>T<br>-<br>i<br>Next(Si)<br>\|S\|<br>C A U G C<br>U A C U C<br>j<br>Next(Sj)<br>\|T\| Case 3: delete: $score(s_i,-)+ALIGN((next(s_i),s_m),(t_k,t_n))$ |

*Table 1: to find the optimal alignment of two suffixes, we need to choose the best of three optimal subalignments: one due to replacement, one due to insertion, and one due to deletion.*

2 I am using replace is used a general term here. If $Si \neq Sj$, then we replaced a base with a different base and score it as a mismatch; if $Si = Sj$, we replace the base with an identical copy and score it as a match.

Putting the three cases together, we see that in $ALIGN((s_i,s_m),(t_k,t_n))$ either $s_i$ replaces $t_k$, $t_k$ is inserted, or $s_i$ is deleted. Therefore to find the optimal score for $ALIGN((s_i,s_m),(t_k,t_n))$, we just have to choose the action from {replace, insert, delete} that has the best score. We then have the recurrrence:

$ALIGN((s_i,s_m),(t_k,t_n))$ = best of
replace $score(s_i,t_k) + ALIGN((next(s_i),s_m), (next(t_k),t_n))$
insert $score(-,t_k) + ALIGN((s_i,s_m), (next(t_k),t_n))$
delete $score(s_i,-) + ALIGN((next(s_i),s_m), (t_k,t_n))$

Since each alignment depends only on three smaller subalignments, we should be able to compute the full alignment $ALIGN((s_1,s_m),(t_1,t_n))$ by starting with the smallest subalignments and gradually building them up. What are the smallest subalignments? Well, they are alignments that include the empty sequence. ALIGN(Ø,Ø) is an empty alignment. An alignment of a suffix with Ø must put gaps against every nucleotide in the suffix, so it only contains insertions or deletions. Hence

$$ALIGN(\text{Ø},\text{Ø}) = 0$$
$$ALIGN((s_i,s_m),\text{Ø}) = \sum_{s_j \in (s_i,\, s_m)} score(s_j, -)$$
$$ALIGN(\text{Ø},(t_k,t_n)) = \sum_{t_l \in (t_k,\, t_n)} score(-, t_l)$$

Now we have our algorithm: we initialize it with the optimal scores for these small subalignments, then use the recurrence $ALIGN$ to compute larger and larger alignments until we finally have ALIGN($(s_1,s_m),(t_1,t_n)$), the optimal alignment of $S$ and $T$.

It is easy to see that the algorithm's space complexity is O($mn$) - it needs to store the subalignments of each of the $m$ suffixes of $S$ with each of the $n$ suffixes of $T$. To compute each subalignment, the algorithm just looks up the scores of three subalignments and does a little addition, so it does O(1) work for each subalignment. Hence the time complexity is also O($mn$).

This algorithm computes the optimal alignment score, but how can we reconstruct the optimal alignment? Well, each alignment is built from an operation on single bases - replace, insert, or delete - and a smaller subalignment. For each subalignment ALIGN((si,sm),(tk,tn)), let us keep a record π((si,sm),(tk,tn)) of the operation we chose to build it. Then we can reconstruct the alignment by traceback through these records. The full alignment's record π((s1,sm),(t1,tn)) tells us which subalignment it was built from, which in turn tells us which subalignment it was built from, and so on. Each traceback step tells us one position in the alignment. We continue this traceback until we reach π(Ø,Ø), at which point we have the full alignment.

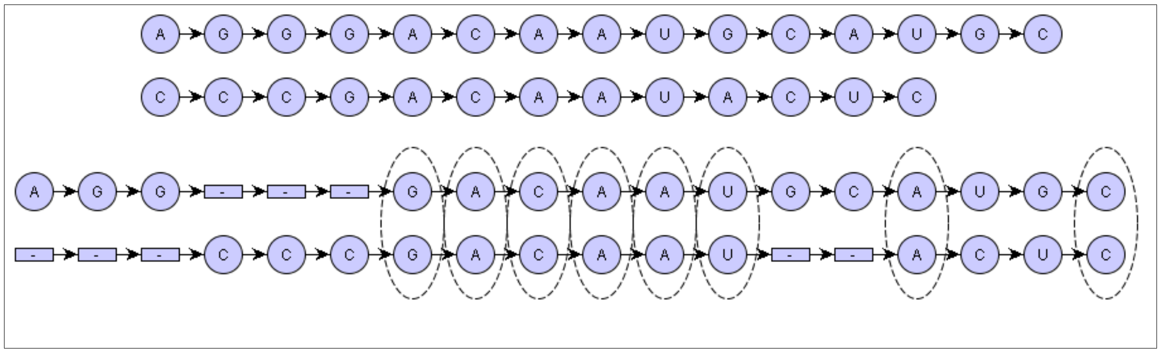

A G G G A C A A U G C A U G C
C C C G A C A A U A C U C
A G G - - - G A C A A U G C A U G C
- - - C C C G A C A A U - - A C U C

*Table 2.: alignment of two sequences. Top: the two sequences. Bottom: their alignment. Matched bases are circled, while mismatched bases are not. Note that gaps have been inserted into each ot the sequences.*

The reason that I've spent so much time explaining a well-known algorithm is that all of the methods I discuss in this thesis are variations on this simple dynamic programming algorithm. We can describe them as follows:

1. Compute the optimal scores for the smallest, trivial subalignments,
2. Use a recurrence to build optimal large alignments from optimal small ones,
    a) while keeping a record of how we built them.
3. Trace back through our records to reconstruct the optimal alignment

In general, I will only explain step 2 in depth. Steps 1, 2a, and 3 are trivial for most of the algorithms,[3] so I will only discuss them briefly.

Note that in the recurrence above, I used a 'best' operator rather than 'min' or 'max'. There are two main goals for aligning two sequences: to minimize the distance between them, or maximize the similarity between them. We calculate the distance between two aligned sequences by setting the match reward to 0 and mismatch and gap scores to be positive. Thus the alignment distance between two sequences is always positive, unless they are perfectly aligned and have zero distance. On the other hand, similarity scores can be either positive or negative; we calculate the similarity between two aligned sequences by setting match scores to be positive and mismatch and gap scores to be negative.

So far we have considered only *global alignment* - aligning the whole sequence *S* with the whole sequence *T*. Both distance and similarity scores are effective in this case. However, if we want to find a *local alignment* which aligns only part of *S* to only part of *T*, then we have to use similarity scores. Why is this? Let us say we are allowed to choose only a subsequence $(s_i,s_j)$ of S to align against a subsequence $(t_k,t_l)$ of T. Then the optimal-distance alignment will always be $(s_i,s_j) = (t_k,t_l) = \emptyset$, since they have zero distance from each other and we can never do better than zero distance. On the other hand, an optimal-

3 Unless we want to improve the space complexity. With many DP algorithms we can find ways to store fewer records or delete unnecessary ones. This isn't my focus here, but the algorithms in this paper could be improved in this way.

similarity alignment will find some tradeoff between bigger and smaller subsequences; bigger subsequences can have more matches, but also more gaps and mismatches. Depending on how we define the similarity score, we can bias it to small regions of very high homology or large regions of moderate homology.

Local alignment is very useful for analysis of biological sequences. After all, we are using alignment to find regions of homology between *S* and *T*, and we may not know where those regions of homology begin. Also, if we want to search a large sequence *S* for a small pattern *T*, then global alignment is likely to give poor results. Therefore I've chosen to develop methods that can be used for both global and local alignment. For sequences, we can do local alignment with just a small modification to the global algorithm (Smith and Waterman, 1981). Since the more complex algorithms I discuss in this thesis can be modified in an exactly analogous way, I will explain it here for this simple case.

We will make two modifications to the global alignment algorithm. First, we initialize the algorithm with ALIGN$((s_i,s_m),Ø)$ = ALIGN$(Ø,(t_k,t_n))$ = 0. This means that all all gaps at the end of the alignment are free of cost. Second, we no longer assume that the optimal alignment includes the full sequences of both S and T. That is, we don't necessarily start our traceback at $\pi((s_1,s_m),(t_1,t_n))$. Instead, we start at the best alignment with at least one full sequence, so that all gaps at the start of an alignment are free of cost. That is, we choose to start our traceback at

$$argmax\left\{\bigcup_{t_k\in T}\{ALIGN((s_1,s_m),(t_k,t_n))\}\cup\bigcup_{s_i\in S}\{ALIGN((s_i,s_m),(t_1,t_n))\}\right.$$

This is a very most commonly used method for local alignment. However, note that it doesn't find the highest-similarity alignment over all substrings of *S* and substrings of *T*. For example, if we align “aaauuuu” with “ggguuuu”, we still have to add three gaps inside the alignment no matter where we place our cost-free start gaps. However, the Smith-Waterman method is generally the right thing to do for phylogenetically related sequences, in which case we assume that *S* and *T* are descendants of a common ancestral sequence *A*. If *S* and *T* are descended from non-overlapping subsequences of *A*, they shouldn't align at all. On the other hand, if *S* and *T* are descended from overlapping subsequences of *A*, then they should only need gaps at the ends for their homologous regions to match.

That said, we can find the optimal alignment of any two substrings in *S* and *T* if we wish. First, we change our recurrence to always maximize over an extra term, *ALIGN*(Ø,Ø) = 0. This lets us replace any score-worsening suffix of the alignment with an empty alignment. Second, we simply start our traceback at the highest scoring subalignment. This lets us ignore any score-worsening prefix of the alignment. Hence our traceback will give us the pair of subsequences in *S* and *T* that have the highest alignment similarity.

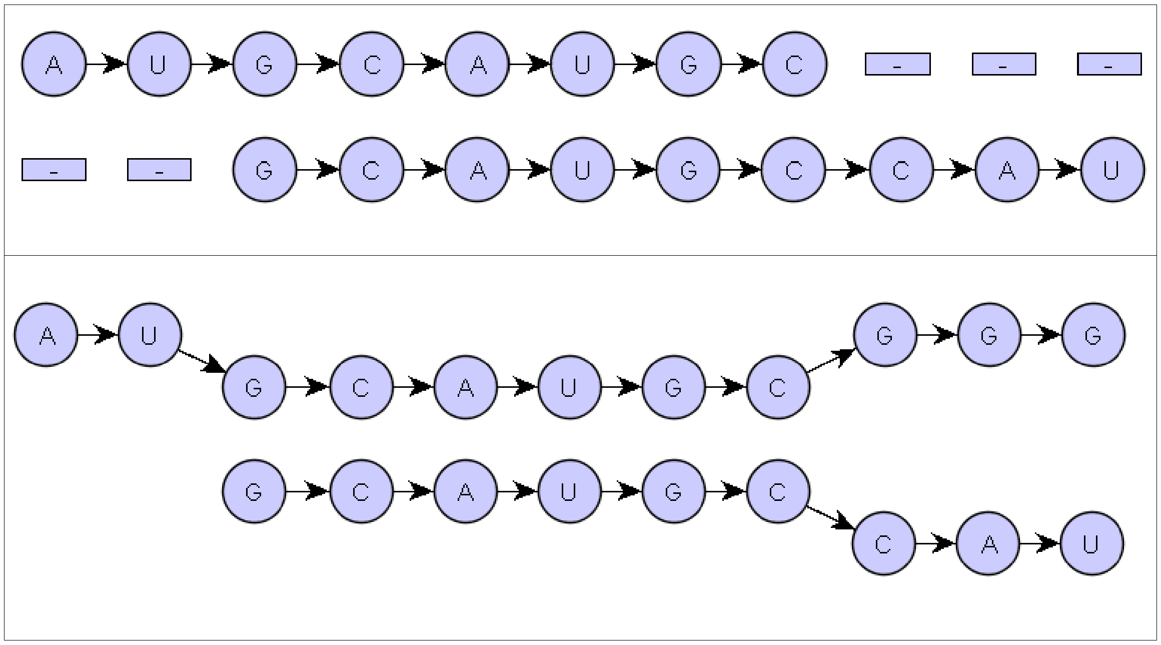


*Table 3.: local sequence alignment. Top: Smith-Waterman local alignment. Because start and end gaps are free of cost, its alignment score doesn't include any gaps. Bottom: alignment of the most-similar pair of subsequences from two sequences. Here the right and left sides the the alignmetn are 'empty', without any gaps at all.*

As I said earlier, the major alignment algorithms that I discuss in this thesis are generalizations of this simple dynamic programming method. It is important to note that there are many methods to solve pairwise sequence alignment with lower complexity bounds, or under more biologically correct scoring schemes for indels, or as fast approximations, and so on. Certainly, the algorithms in this thesis can also make use of some of these augmentations, and I will consider a few interesting ones in the 'Discussion' section. For example, all of the algorithms in this paper can be modified to use the popular 'affine' gap costs, in which the cost of an indel is ($ax + b$), where $x$ is the number of gaps, $a$ is the penalty for each additional gap, and $b$ is a penalty for 'opening' the gap in the first place. This is a better approximation of natural evolutionary processes in which insertion and deletion mutations are rare but have variable sizes. There are many other models for scoring indels that are more biologically motivated (e.g. logarithmic gap penalties suggested by (Gu and Li, 1995)) which might also be applicable to the algorithms developed in this thesis. However, my main focus in this thesis is partial ordering, not scoring schemes. I will mostly consider alignment under a simple scoring model with unit-cost gaps.

### 2.2 Progressive profile alignment of multiple sequences

So far we have considered alignments of two sequences, commonly known as *pairwise* alignments. Now, let us consider the problem of aligning multiple sequences. A row-column multiple sequence alignment is just like a pairwise alignment, but with many sequences in a stack:

```
S1 = aug---gacugug-----
S2 = agg---gacaaugcaugc
S3 = ---cccgacaau--acuc
```

Qualitatively, we can see that this alignment does a good job of matching similar regions of $S_1$, $S_2$, and $S_3$. We need to quantify this goodness with a scoring function before we can write an algorithm for multiple alignment. Although there is no general agreement on a 'best' type of multiple alignment score [cite], the most popular one is *sum-of-pairs* [cite]. The sum of pairs score is just the sum of the scores of all pairwise alignments that are contained in the multiple alignment. For example, the alignment above would have a score equal to the sum of the scores for these pairs:

```
S1 = aug---gacugug-----
S2 = agg---gacaaugcauuc

S2 = agg---gacaaugcaugc
S3 = ---cccgacaau--acuc

S1 = aug---gacugug-----
S3 = ---cccgacaau--acuc
```

In order to make these into valid pairwise alignments, we ignore all of the columns that only have gaps. Although it is simple to calculate the sum-of-pairs score of an alignment, it is NP-Complete to find an alignment with the optimal sum-of-pairs score under the scoring model we've considered (Wang and Jiang, 1994). This is unfortunate, because multiple alignments are bread and butter for studies of molecular evolution. Multiple alignments allows us to find evolutionary relationships between DNA, RNA, and protein sequences, to build models for recognizing and analyzing families of genes, and to gain insight into their biological roles. For example, positions of a multiple alignment that are the same for many related sequences are probably crucial for that molecule to function properly. Therefore multiple alignment has been the subject of unrelenting, intense study; there are many, many approximation algorithms for this problem.

The algorithms I will discuss in this paper are based on variations of *progressive* *alignment*.[4] Progressive alignment algorithms build a multiple alignment from a series of pairwise alignments. Of course, this means that our previous *ALIGN*(sequence,sequence) algorithm isn't enough; it outputs pairwise alignments but only takes sequences as input, so we can never produce an alignment with more than two sequences. Therefore we have to define an *ALIGN*(alignment,alignment) algorithm which takes two alignments as inputs and merges them together into a single, larger alignment.

The earliest method of this type is due to (Feng and Doolittle, 1987). Rather than trying to align two entire alignments, they compressed alignments into *alignment profiles*. An alignment profile just keeps track of the number of nucleotides at each position. For example, the alignment written above would be squished into a profile as follows:

```
Alignment:
          aug---gacugug-----
          agg---gacaaugcaugc
          ---cccgacaau--acuc
Profile:
a count = 2      3 22   2
u count =  1       1 3   11
g count =  12   3   1 2   1
c count =    111  3    1 1 2
- count = 111222      121111
```

This is lossy compression, since the profile doesn't have any information about which bases follow which. For example, this profile would be consistent with an alignment that has a sequence starting in “augccc”. However, this profile representation allows us to use our *ALIGN*(sequence,sequence) algorithm to align profiles with only a slight modification. All we need to do is define a scoring function *score*(*p,p*') for aligning two profile positions *p* and *p*', where *p* and *p*' are in {a,u,g,c,-} x $\Re$ .[5] For example, we might use a sum-of-pairs style score by setting *score*(*p*,*p*') as follows:

$$score(p,p') = \sum_{\substack{b \in p \\ b' \in p'}} score(b,b')(\textit{fraction of } b \textit{ in } p)(\textit{fraction of } b' \textit{ in } p')$$

An important note is that since sum-of-pairs multiple alignment is NP-complete, *ALIGN*(alignment,alignment) can only approximate an optimal alignment, even if it is based on a more faithful representation than profiles. These approximations are generally much better if the alignments they merge are relatively similar to each other. Intuitively, we don't have to insert many gaps if the alignments are similar, so there are fewer ways to

4 'Progressive alignment' is sometimes used to refer only to methods guided by phylogenetic trees. Here, I will use the term to refer to any algorithm for building multiple alignments from a series of pairwise 'alignments of alignments'.

5 That is, a profile position is a set of pairs of (character, character count).

insert them poorly. Therefore the the order we choose for merging the multiple alignments is very important. If we simply merge each sequence into a multiple alignment one by one, we will get very bad results.

(Feng and Doolittle, 1987) and many others since then have used phylogenetic trees to guide the multiple alignment. A phylogenetic tree represents the evolutionary relationships of a set of species. Each leaf on the tree is a sequence, and each internal node represents the most recent common ancestor of all the sequences in its subtree. Therefore, the more closely related the sequences, the closer to each other they are on the tree. This is why phylogenetic guide trees are so popular for this purpose; a good phylogenetic guide tree greatly improves the quality of an alignment. There are many methods for computing a phylogenetic tree for *n* sequences from an *n* by *n* matrix of the distance between each pair (e.g. Fitch and Margoliash, 1967; Saitou and Nei, 1987), which we can compute using pairwise alignments. With the tree in hand, we now build a multiple alignment by successively merging the two children of each internal node.

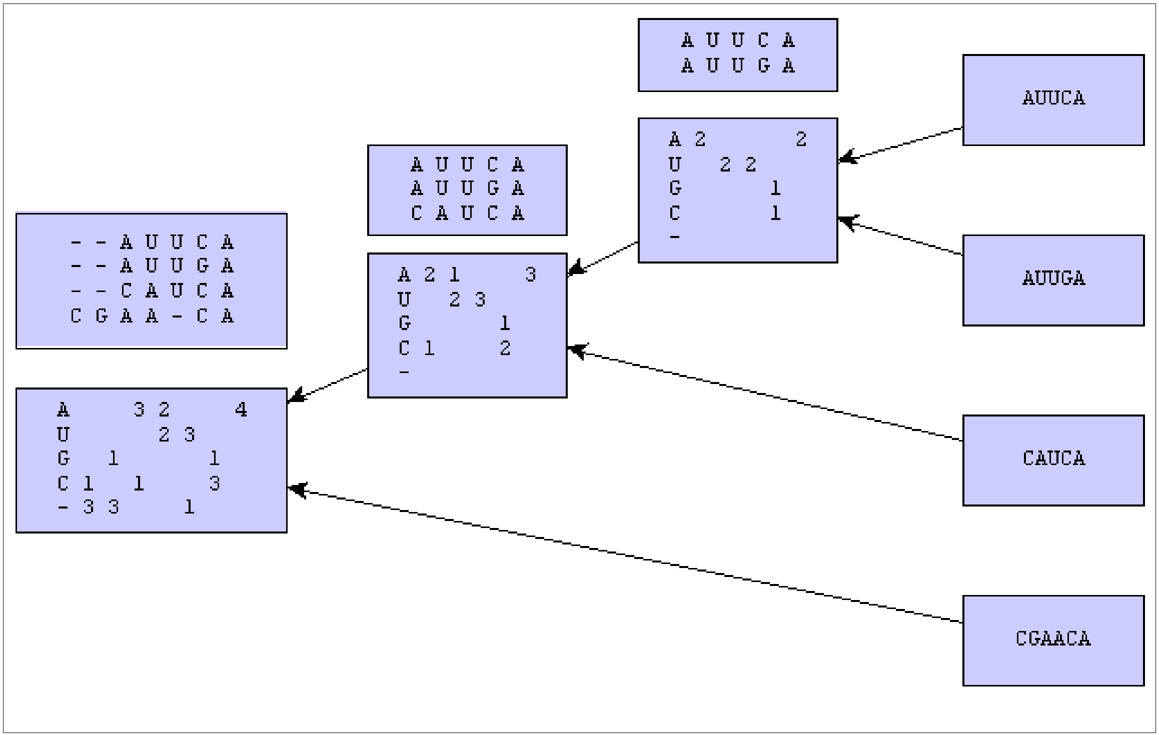


*Table 4.: progressive profile alignment guided by a phylogenetic tree. Alignment profiles are drawn on the nodes of the tree. Alignments are drawn above the profiles. Ideally, each successive alignment merges the two most similar sequences.*

If we have *k* sequences to align of length at most *n*, this method calls *ALIGN*(sequence,sequence) *k*(*k* - 1)/2 times to compute the matrix of pairwise distances, then calls some combination of *ALIGN*(sequence,sequence) and *ALIGN*(alignment,alignment) to compute alignments for the ($k$ - 1) internal nodes of the tree. Hence the algorithm runs in O($n^2k^2$) time and O($n^2$) space.[6] This is not actually true; each alignment will insert some gaps into the sequences, and once inserted, gaps can never be removed. Hence each merged alignment will be longer than the alignments from which it was generated. We can imagine a nightmare scenario in which each alignment contains as many gaps as possible, so each profile for $k$ sequences has O($kn$) positions, greatly increasing the method's running time. However, we are using this method to align sequences with some significant evolutionary relationships to each other, so they are not too dissimilar. Typically sequences with very low similarity are simply not aligned or are processed separately from the rest (e.g. Thompson *et al.*, 1994). This sort of issue is not my main concern in this thesis, although I will touch on it briefly.

## *2.3 Partial-order alignment of sequences*

As I alluded to in the previous section, there are several troubling aspects of the traditional row-column and row-column profile representations for multiple alignments. Each sequence is filled with 'gap' characters that don't represent any of its actual features, but rather its relationships to other sequences. Shouldn't we represent these relationships with some kind of direct mapping, rather than manipulating the sequences themselves? However, a row-column profile doesn't have any information about a mapping between sequences. For example, let us consider two different sequences we might to the alignment that we examined in the previous section. In the first case, we'll just add another copy of $S_2$; in the second case, we'll add a new sequence $S_4$.

| | |
|---|---|
| $S_1$ = `aug---gacugug-----` | $S_1$ = `aug---gacugug-----` |
| $S_2$ = `agg---gacaaugcaugc` | $S_2$ = `agg---gacaaugcaugc` |
| $S_3$ = `---cccgacaau--acuc` | $S_3$ = `---cccgacaau--acuc` |
| $S_2$ = `agg---gacugugcaugc` | $S_4$ = `agg---gacugugcacgc` |

Even though $S_2$ is already in the alignment and $S_4$ has a subsequence ("cacgc") that never appeared in any of the original sequences, they both receive an equal alignment score. We could avoid this problem by guiding the alignment with a phylogenetic tree so that the identical sequences are aligned together before any gaps or mismatches can pollute the profile. But it still seems like a strange order dependence when an alignment can't recognize when it has a perfect template for matching a new sequence unless the alignment was built in a correct order.

In addition, gaps are a highly degenerate representation for indels. We only insert gaps in order to maximize the match/mismatch scores, but there are combinatorically many ways to insert gap characters and still have exactly the same matches and

6 Plus O($n|S|$) space to store the alignment, but typically $|S| << n$, so the O($n^2$) factor is the bottleneck.

mismatches. Yet the choice of gap insertions has a large effect on alignment. We can generate two equally scoring alignments which represent the same sequence of insertions and deletions:

| | |
|---|---|
| $S_1$ = aug---gacugug----- | $S_1$ = ---auggacugug----- |
| $S_2$ = agg---gacaaugcaugc | $S_2$ = ---agggacaaugcaugc |
| $S_3$ = ---cccgacaau--acuc | $S_3$ = ccc---gacaau--acuc |

But if we align a sequence “ggcgacaaugc” with these, we get very different results:

| | |
|---|---|
| $S_1$ = aug---gacugug----- | $S_1$ = ----auggacugug----- |
| $S_2$ = agg---gacaaugcaugc | $S_2$ = ----agggacaaugcaugc |
| $S_3$ = ---cccgacaau--acuc | $S_3$ = -ccc---gacaau--acuc |
| $S_4$ = --gcc-gacaaugc---- | $S_4$ = gcc----gacaaugc---- |

(Lee *et al.*, 2002) argue that the root of these problems is that a row-column alignment forces a total ordering on its positions. That is, each position in the alignment has one predecessor and one successor.[7] However, even with an optimal pairwise alignment, we can only deduce a partial ordering. Based on the information in $S_1$, $S_2$, and $S_3$, we simply can't know whether “agg” comes 'before' or 'after' “ccc” in some ancestral sequence, or if they even ever coexisted in a sequence. Yet row-column alignments assume that “agg” and “ccc” do have some ordering.

To avoid the difficulties of gapped, total-order methods, (Lee *et al.*, 2002) introduced *partial-order multiple sequence alignments* (PO-MSA). A partial order alignment is a directed acyclic graph. Each node in the PO-MSA is a position in the alignment. If the graph were constrained to have a total ordering, each node would have at most one incoming edge (predecessor) and one outgoing edge (successor). However, in a PO-MSA, each node can have multiple predecessors and multiple successors. This allows the PO-MSA to preserve all of the information about each of its sequences; each sequence that has been merged into a PO-MSA still exists as a subgraph of that PO-MSA. Therefore when we align a sequence *S* with a PO-MSA *T*, we implicitly consider the optimal alignment of *S* with every sequence in *T*. See the following figure for an illustration of PO-MSA in action.

The idea behind the PO-MSA algorithm is simple. In total-order sequence alignment, each subsequence ($s_i$,$s_m$) that we consider has exactly one 'next' suffix, but in a PO-MSA, each node $s_i$ has one suffix for each outgoing edge. So to find an optimal alignment, instead of maximizing over a single 'next' suffix, we maximize over several different 'next' suffixes. That's all there is to it; we can use exactly the same algorithms as we do in ordinary sequence alignment. The key step is how we merge together two

7 To be technically correct, the first position has no predecessor and the last position has no successor.

aligned sequences into a partial order graph.

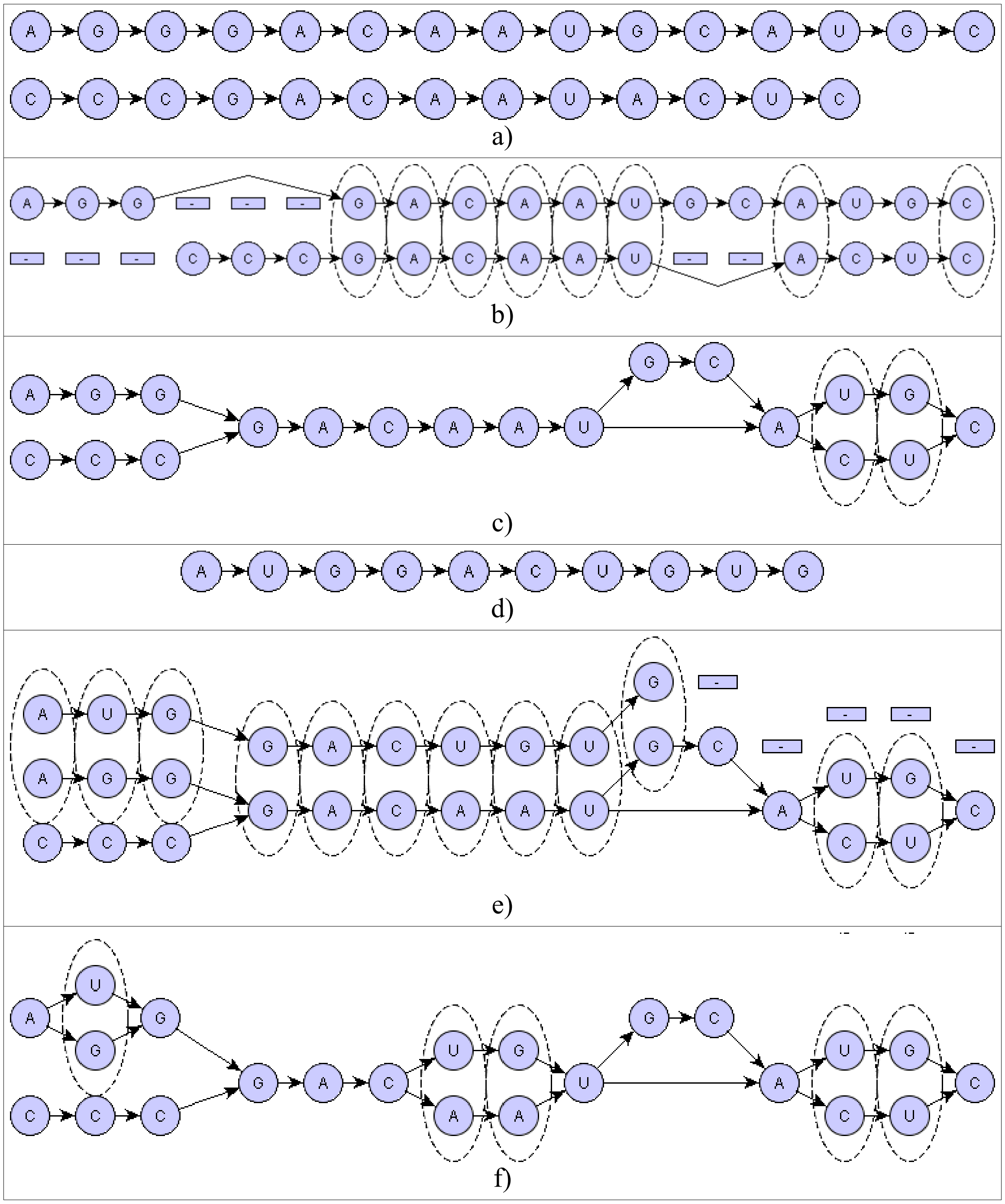


*Table 5: partial order alignment of sequences. a) two sequences represented as graphs. b) Ordinary sequence alignment with gaps. c) node fusion. Note that each sequence is a subgraph of the PO-MSA. d) a*

*third sequence. e) alignment of the sequence with the PO-MSA. The alignment considers all paths through the PO-MSA and aligns the sequence with the optimal one. f) node fusion into a new PO-MSA .*

I will now give a more formal presentation of the algorithm *ALIGN*(PO-MSA,PO-MSA) (as in Grasso and Lee, 2004). Recall that for two suffixes $(s_i,s_m)$ and $(t_k,t_n)$ of sequences $S$ and $T$, the standard algorithm's recurrence was:

$ALIGN((s_i,s_m),(t_k,t_n))$ = best of
replace = $score(s_i,t_k) + ALIGN((next(s_i),s_m), (next(t_k),t_n))$
insert = $score(-,t_k) + ALIGN((s_i,s_m), (next(t_k),t_n))$
delete = $score(s_i,-) + ALIGN((next(s_i),s_m), (t_k,t_n))$

For two PO-MSAs, however, each node can have more than one next suffix. That is, *next*(*n*) is not a single successor node, but rather a set of successor nodes. To find the optimal alignment, we simply maximize over all possible pairs of next suffixes:[8]

$$ALIGN((s_i,s_m),(t_k,t_n)) =$$
$$\max\begin{cases} replace = score(s_i,t_k) + \max\limits_{\substack{s_{next}\in next(s_i)\\ t_{next}\in next(t_k)}} ALIGN((s_{next},s_m),(t_{next},t_n)) \\ insert = score(-,t_k) + \max\limits_{t_{next}\in next(t_k)} ALIGN((s_i,s_m),(t_{next},t_n)) \\ delete = score(s_i,-) + \max\limits_{s_{next}\in next(s_i)} ALIGN((s_{next},s_m),(t_k,t_n)) \end{cases}$$

Note that the notation $score(-,t_j)$ just means the score for an insertion of $t_j$; we don't actually introduce gaps into the alignment.

This recurrence make sense on the face of it, but what is a 'suffix' of a partial order alignment? A row-column profile has a clear 'start' and 'end' because the positions have a total ordering. But a PO-MSA represents many different sequences, each of which could have different starting and ending nodes. Therefore a PO-MSA $S$ has to keep lists of the start nodes $S_{start}$ and end nodes $S_{end}$ for each sequence in the alignment. To do global alignment, we'll find the optimal alignment between any one sequence in $S$ and any one sequence in $T$. We'll do this by maximizing over all pairs of starting and ending nodes:

$$ALIGN(S,T) = \max_{\substack{s_i\in S_{start}\\ s_m\in S_{end}\\ t_k\in T_{start}\\ t_n\in T_{end}}} \left\{ ALIGN((s_i,s_m),(t_k,t_n)) \right.$$

However, we don't actually have to compute $O(|S_{start}||S_{end}||T_{start}||T_{end}|)$ alignments! We can see this by representing $S_{start}$ and $S_{end}$ with nodes in the PO-MSA. $S_{start}$ is a

8 (Lee *et al.*, 2002) and (Grasso and Lee, 2004) describe POA in terms of prefixes rather than suffixes.

predecessor of each start node, and $S_{end}$ is a successor of every end node. Then a global alignment is just ALIGN(($S_{start}$,$S_{end}$),($T_{start}$,$T_{end}$)).

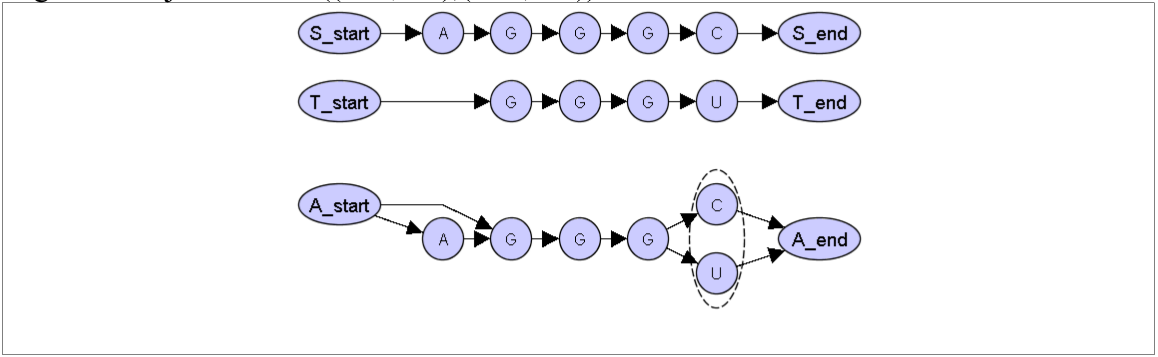


*Table 6: Alignment of PO-MSAs with 'start' and 'end' nodes. Top: sequence S. Middle: sequence T. Bottom: their alignment A. A global alignment with A would be an alignment with ($A_{start}$,$A_{end}$).*

To compute an optimal local alignment, we use the same approach we used for total order sequence alignment. We initialize the alignment so that ALIGN(($s_i$,$S_{end}$),Ø) = ALIGN(Ø,($t_k$,$T_{end}$)) = 0 for all $s_i$ and $t_k$., which makes 'end gaps' cost nothing. We then start the traceback at the subalignment with the highest similarity score, which makes 'start gaps' cost nothing.

Note that this algorithm does not find an optimal alignment of each path in *S* with each path in *T*. Rather, it finds the optimal alignment of any one path in *S* with any one path in *T*. This is a somewhat different notion than row-column profile alignments, which try to match the entirety of each profile. We can think of the PO-MSA algorithm as a pattern matching procedure something like "is any path is *S* like any path in *T*", whereas row-column profile alignment is more like "are all the paths combined in *S* like all the paths combined in *T*".

Once we've found the optimal alignment of *S* and *T*, we merge them together into a new alignment *A*. We do this by processing each aligned pair of nodes ($s_i$,$t_k$) as follows:

1. If $s_i$ and $t_k$ are matched, merge them into a new node $a_{ik}$. The new node has all of the incoming and outgoing edges from both $s_i$ and $t_k$. Remove any redundant edges.
2. If $s_i$ and $t_k$ are mismatched:
    1. If neither $s_i$ nor $t_k$ have been mismatched before, put them into a set of nodes $aligned_{ik}$ that are aligned at the same 'position' in *A* (drawn as dotted circles in the diagrams above).
    2. If either $s_i$ or $t_k$ has been mismatched before, then combine their sets $aligned_i$ and $aligned_k$ into a single set of aligned nodes $aligned_{ik}$. Then merge any nodes in the set that have the same nucleotide.
3. If $s_i$ or $t_k$ is a gap, do nothing.

The first step joins the two PO-MSAs at every one of their matched positions.

This corresponds to an assumption that matched bases are homologous, so paths through one of the bases are equivalent to paths through the other matched base. The second step doesn't merge mismatched nodes; it preserves them as alternate paths through the PO-MSA. However, we do remember which nodes were aligned together using *aligned* sets (indicated by dashed circles in the illustration). We treat node alignment according to a transitive property: if $node_1$ is aligned with $node_2$ and $node_2$ is aligned with $node_3$, then $node_1$ is aligned with $node_3$. Thus, if we align a 'U' with a 'G' and the 'G' had been aligned with another 'U' node in the past, then we merge the two 'U' nodes.[9] In addition, the aligned node sets allow us to translate the PO-MSA into a row-column alignment. Otherwise, we wouldn't know which alternate paths are indels and which are due to mismatches. Finally, the third merging step leaves gapped nodes alone - they will end up as alternate paths through the graph.

At the end of this merging, each of the old PO-MSAs is a subgraph of the new PO-MSA. Note that this merging avoids the degeneracies of 'gapped' representations. A PO-MSA can be translated into a large number of equivalent row-column alignments. This property makes partial-order alignment resistant to order-of-alignment dependencies. Indeed, the first version of PO-MSA simply merged sequences into the alignment one by one without calculating a guide tree. There is still some order dependence,[10] and a tree-guided method improves the results (Grasso and Lee, 2004).

What is the complexity of PO-MSA? If we just do a pairwise alignment of two sequences, its complexity is the same as ordinary sequence alignment. However, if the number of branches is high, we have to do more operations. The most costly step is computing *replace*, becaue we have to optimize over all $s_{next}$ in $next(s_i)$ and all $t_{next}$ in $next(t_k)$. Let $|S|$ be the number of nodes in $S$ and $Edges(S)$ be the number of edges in $S$. We have to store alignment scores for each pair of suffixes, so we need $O(|S||T|)$ space. Note that for each pair of nodes $(s_i,t_k)$, we look up an *ALIGN* score exactly once for each pair of next nodes $(s_{next}, t_{next})$. That is, over the course of the whole alignment, we test each edge in *S* once against each edge in *T*. The number of *insert* and *delete* lookups at each step are fewer than the number of *replace* lookups, so they don't increase the complexity. Hence the pairwise alignment time is $O(Edges(S)Edges(T))$.

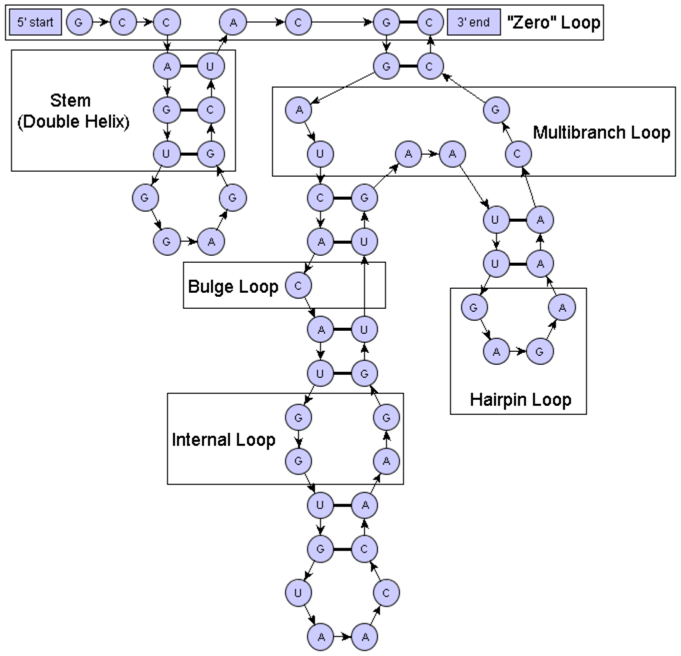
As we can see in the illustration, this merging step compresses a PO-MSA down to a small number of nodes. The more closely related the sequences are, the more compact the graph. We can contrast this to row-column profile alignments, in which every sequence that is aligned adds another row to the multiple alignment and in which every profile position has five separate counts that factor into its scoring function. The compact nature of PO-MSAs results in impressive alignment speeds. (Grasso and Lee, 2004) note that for sequences with high similarity, the program `POA2` comes close to the speed of the `MAFFT` alignment program (Katoh *et al.*, 2002), even though `POA2` computes optimal alignments in quadratic time and `MAFFT` uses fast fourier transforms to do approximate sequence alignments in log-linear time.

9 In `POA2` (Grasso and Lee, 2004), this behavior is optional; it is activated with a "-fuse_all" switch.
10 See section 3.3 for a more detailed discussion.

## 2.4 Basics of RNA structure

DNA and RNA molecules are very similar to each other. Both are sequences (aka strands) composed of nucleotides (aka bases) with adenine, cytosine, and guanine residues ('a', 'c', and 'g'); DNA also has thymine ('t'), whereas RNA has the similar nucleotide uracil ('u'). In both DNA and RNA, some bases pair to each other using hydrogen bonds: *G* to C and A to U. Less stable G-T and G-U 'wobble' pairs are also common. Most other combinations are possible but they are very unstable and hence rare.

However, while DNA molecules are generally locked into long, double-stranded helices, single-stranded RNAs can take on a rich set of structural features that have a great influence on their roles in the cell. The schematic below illustrates some common terms for RNA structural features.



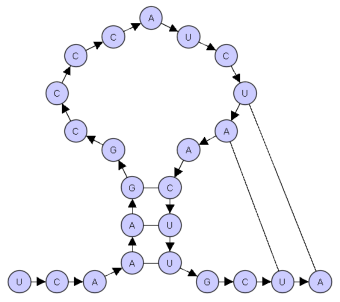
```
...(((....)))..((..((.((..((....))..)))))..((....))..))
GCCAGUGGAGGCUACGGAUCACAUGGUGUAACCAAGGUUGAAUUGAGAAACGCC
```

*Table 7: a hypothetical RNA secondary structure. Top: structure drawn as a typical schematic. Solid lines are bonds between bases. Arrowed lines are the molecule's phosphate backbone, starting at the 5' side and ending at the 3' side. Structures are often described in terms of stems - series of paired bases that form a double helix - and loops - sequences of unpaired bases. The 'zero' loop is formed by the start and end of the sequence. A hairpin loop touches only one stem. An internal loop has two sides, each touching two different stems. A bulge loop has one side with zero bases. A multibranch loop touches more than two stems; a series of connected multibranch loops give rise to complex shapes with many branches. Bottom: the structure in 'bracket' notation. '.' is an unpaired base, '(' is a bases paired toward downstream, toward the 3' end of the structure, and ')' is a base paired upstream, toward the 5' start of the structure.*

This thesis is about algorithms for comparing RNA structures. In order to compare them, we'll first have to choose data structures to represent them. As a first step, I will consider some theoretical and practical restrictions we can impose on RNA structures.

There are a few theoretical constraints on the number of nucleotides in RNA loops. The phosphate backbone of an RNA strand has a physical limit on its bond angles, so the molecule cannot make sharp twists. Therefore RNAs cannot form hairpin loops of fewer than three bases. Also, although there is no hard upper limit on loop sizes, their sizes are not completely arbitrary. This is due to combinatorics and thermodynamics: a long enough stretch of a nucleotides is bound to have some potential pairings, and base pairs are thermodynamically favorable. For this reason, many algorithms on RNA structures limit their consideration to loops of size less than 30 bases or so (e.g., see Lyngsø *et al.*, 1999), which is still rather large, since loops are typically 10 or fewer bases. Although the algorithms that I build on and develop in this thesis do not require limits on the size of loops, some of them will run more efficiently for structures with small loops.

In computational studies, it is also common to consider only RNA *secondary structures*, rather than their full *tertiary structures*. A secondary structure is one in which all base pairs are nested; no base pairs are crossing. To be formal: let us denote a pair from the $i$-th base to the $j$-th base as $(i,j)$, for $i < j$. An RNA's secondary structure consists of all base pairs such that for any $(i,j)$ and $(u,v)$, if $i < u$, then $v < j$. The RNA's tertiary structure consists of all other base-pairing interactions, as well as a some exotic interactions such as base triplets.



```
...(((........[[.)))..]]
```

*Table 8: a simple tertiary structure. Top: a H-type pseudoknot (so-called because it is formed on a hairpin loop). Bottom: bracket notation for tertiary structure. Ordinary bracket notation can only represent nested base pairs. Here, square brackets denote non-nested base pairs that 'cross' the secondary structure.*

Tertiary structure is sometimes left out of consideration because it is computationally hard to work with. For example, it is NP-hard to compute the energetically optimal tertiary structure of an RNA (Akutsu 2000; Deogun *et al.* 2004) or to compute the optimal edit distance between two tertiary structures under certain editing models (Wang and Zhang, 2001), whereas both of these problems can be solved in polynomial time for secondary structures (Zuker *et al.*, 1999; Bafna *et al.*, 1996). Most of this thesis is devoted to optimal comparisons between of secondary structures. However, following (Wang and Zhang, 2001), I will develop an algorithm that builds approximate tertiary comparisons from a series of secondary structure comparisons.

There are quite a few classes of algorithms for the study of RNAs. There are many methods for predicting the structure of RNA sequences based on thermodynamics or evolutionary information, for scanning genomes for RNAs that can fold into a particular structure, and even for simply drawing RNA structures without overlapping lines. Even within the class of RNA structure comparison algorithms, there are a large number of methods and representations for RNA structures. In this thesis I will only address the problem of *aligning multiple RNA structures*: establishing a correspondence between structural features so that we can overlay a set of structures upon each other.

## 2.5 Alignment of two RNA secondary structures

### 2.5.1 Pairwise alignment of arc-annotated strings

(Bafna *et al.*, 1996) represented and aligned RNA structures as *arc-annotated strings*. An arc-annotated string $S$ is a sequence of $m$ nucleotides $S_s = (s_1,s_2,...,s_m)$ and a set of $m_p$ base-pairs $S_p = \{(s_i,s_j),(s_k,s_l),...\}$. The main idea is to constrain a sequence alignment to take into account base pairing interactions.

The algorithm of (Bafna *et al.*, 1996) for aligning two arc-annotated strings $S$ and $T$ has to align each substring of $S_s$ with each substring of $T_s$, so it has time and space complexity $O(m^2n^2)$. (Wang and Zhang, 2001) showed that certain substring-substring alignments can be omitted, which reduces the time and space complexity to $O(mm_pnn_p)$. In general $O(m_p) = O(m)$, but this is still an improvement because there are always fewer pairs than bases, and typically $m_p < (m / 3)$. Here I will present the method of (Wang and Zhang, 2001).[11]

11 For simplicity, I will present the algorithm using unit-cost gap penalties. (Wang and Zhang, 2001) show how to extend the algorithm to use affine gap penalties, which is relatively straightforward.

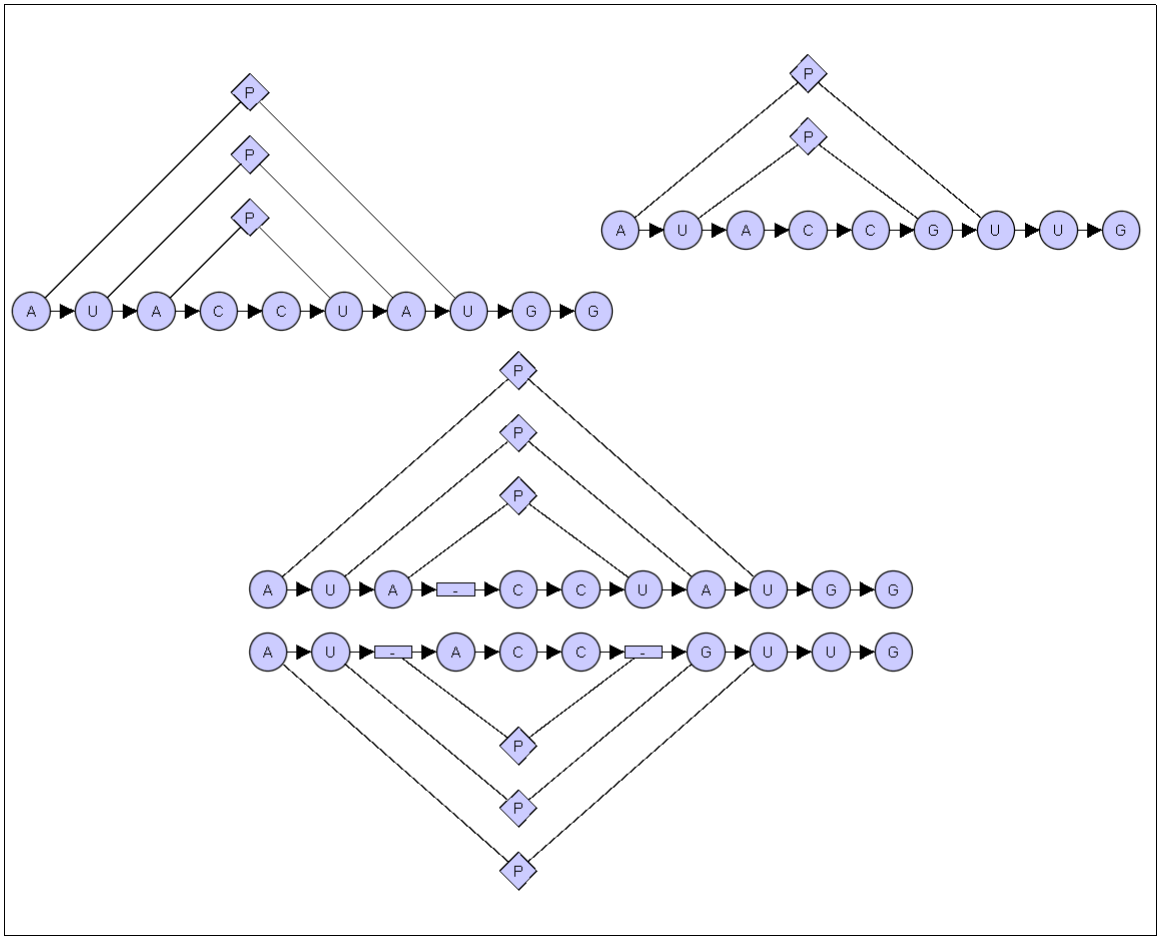


*Table 9: arc-annotated strings. Top: arc-annotated strings S (left) and T (right). Bottom: alignment of the two. Note that to satisfy the arc-constraint, a base pair must be aligned as a whole unit: either it is aligned with another base pair, or it is aligned with a 'gap' base pair. The alignment has only a single set of arcs that is imposed on both structures.*

The algorithm basically forces a sequence alignment to satisfy an arc-matching constraint: if $(s_i,s_j)$ is in $S_p$, then we can only align $s_i$ with $t_k$ if $(t_k,t_l)$ is in $T_p$ and we align $s_j$ with $s_l$. That is, $(s_i,s_j)$ is treated as a whole unit rather than as two individual bases. This means that the resulting arc-annotated alignment has only a single set of arcs, and hence represents only a single consensus structure. See the table above for an example of an arc-annotated alignment.

First, we'll need some notation. For an arc-annotated string $S$ with sequence $S_s$ and pairs $S_p$:

$(s_i,s_j)$ is a subsequence containing all of the nucleotides from $s_i$ to $s_j$.

$next(s_i) = s_{i+1}$, or Ø if $i = m$.

$prev(s_i) = s_{i-1}$, or Ø if $i = 1$.
$pair(s_i) = (s_i,s_j)$ or $(s_j,s_i)$ if either base pair is in $S_p$, or Ø if si is unpaired.
$base\_score(s_i,t_k)$ is the score for aligning $s_i$ with $t_k$.
$pair\_score((s_i,s_j),(t_k,t_l))$ is the score for aligning the pair $(s_i,s_j)$ with the pair $(t_k,t_l)$.
$(s_i,s_j)$ is a *suffix* if $next(s_j) =$ Ø
$(s_i,s_j)$ is a *pair-suffix* of base pair $p = (s_a,s_b)$ if $a < i$ and $next(s_j) = s_b$.

For example, consider the the arc-annotated string $S$ in the table above. “auaccuaugg” and “augg” are suffixes of $S_s$. “cc” and “c” are pair-suffixes of the innermost base pair. “uaccua” and “cua” are pair-suffixes of the outermost base pair.

As in sequence alignment, the optimal score for a global alignment will the the score computed for $ALIGN((s_1,s_m),(t_1,t_n))$. Let us now consider how we can break each $ALIGN((s_i,s_j),(t_k,t_l))$ into subalignments. If $s_i$ is not paired and $t_k$ is not paired, then we proceed as we would in sequence alignment:

```
if $pair(s_i) = pair(t_k) =$ Ø:
$ALIGN((s_i,s_j),(t_k,t_l))$ = max of
replace    $base\_score(s_i,t_k)$ + ALIGN($(next(s_i),s_j), (next(t_k),t_l)$)
insert     $base\_score(-,t_k)$ + ALIGN($(s_i,s_j), (next(t_k),t_l)$)
delete     $base\_score(s_i,-)$ + ALIGN($(next(s_i),s_j),(t_k,t_l)$)
```

If $s_i$ is paired but $t_k$ is not paired (i.e. $(s_i,s_m)$ or $(s_m,s_i)$ is in $S_p$), we must place a gap against either $s_i$ or $t_k$. Otherwise, we would break the arc-constraint. If we place a gap opposite $s_i$, we'll charge it half of the cost for gapping $(s_i,s_m)$. Why only half the cost? Since we gapped $s_i$, the arc-matching constraint will force the remaining subalignment, $ALIGN((next(s_i),s_j), (t_k,t_l))$, to gap $s_m$ (as we'll see later on). When it does, we'll charge it the other half of the cost. Thus the whole alignment includes the total cost for gapping the base pair $(s_i,s_m)$.

```
if $pair(s_i) = (s_i,s_m)$ and $pair(t_k) =$ Ø:
$ALIGN((s_i,s_j),(t_k,t_l))$ = max of
insert     $base\_score(-,t_k) + ALIGN((s_i,s_j), (next(t_k),t_l))$
delete     $(1/2) * pair\_score((s_i,s_m),(-,-)) + ALIGN((next(s_i),s_j), (t_k,t_l))$
```

If $s_i$ is unpaired but $t_k$ is paired, it is a mirror case of the above.

What if both $s_i$ and $t_k$ are paired, i.e. $(s_i,s_m)$ is in $S_p$ and $(t_k,t_n)$ is in $T_p$? There are two cases. First, if $s_m$ is not in the subsequence $(s_i,s_j)$ or $t_n$ is not in the subsequence $(t_k,t_l)$, then we can't align those base pairs; they are incompatible with the substrings $(s_i,s_j)$ and $(t_k,t_l)$. This can happen if $s_i$ is the second base in its pair, so $s_m$ occurs before $(s_i,s_j)$ - that is, if the current alignment is part of a large alignment that will gap $s_m$. This can also happen if the arc-annotated string has tertiary structure (explained in more detail below). In either case, we must again gap $s_i$ or $t_k$.

if $pair(s_i) = (s_i,s_m)$ or $pair(t_k) = (t_k,t_n)$ is incompatible with $(s_i,s_j)$ or $(t_k,t_l)$:
$ALIGN((s_i,s_j),(t_k,t_l))$ = max of

| | |
|---|---|
| insert | $(1/2)*pair\_score((-,-),(t_k,t_n)) + ALIGN((s_i,s_j), (next(t_k),t_l))$ |
| delete | $(1/2)*pair\_score((s_i,s_m),(-,-)) + ALIGN((next(s_i),s_j), (t_k,t_l))$ |

In the second case, $(s_i,s_m)$ is inside $(s_i,s_j)$ and $(t_k,t_n)$ is inside $(t_k,t_l)$, so we can align the two base-pairs. Then the optimal alignment constrained by that base pair will include two subalignments. The first is an alignment of the 'loops' defined by the base pairs - the nucleotides between the paired bases $(s_i,s_m)$ and between $(t_k,t_n)$. Hence we must align $(next(s_i),prev(s_m))$ with $(next(t_k),prev(t_n))$. The second is an alignment of the rest of the bases that are left over - we must align $(next(s_m),s_j)$ with $(next(t_n),tl)$.

if $pair(s_i) = (s_i,s_m)$ and $pair(t_k) = (t_k,t_n)$, and they are compatible with $(s_i,s_j)$ and $(t_k,t_l)$:
$ALIGN((s_i,s_j),(t_k,t_l))$ = max of

| | |
|---|---|
| replace | $pair\_score((s_i,s_m),(t_k,t_n))$ $+ ALIGN((next(s_i),prev(s_m)),(next(t_k),prev(t_n)))$ $+ ALIGN((next(s_m),s_j),(next(t_n),tl))$ |
| insert | $(1/2)*pair\_score((-,-),(t_k,t_n)) + ALIGN((s_i,s_j), (next(t_k),t_l))$ |
| delete | $(1/2)*pair\_score((s_i,s_m),(-,-)) + ALIGN((next(s_i),s_j), (t_k,t_l))$ |

To complete these recurrences, we have to define what happens when we have an empty subsequence, i.e. $(s_i,s_j)$ where $s_j$ comes before $s_i$ in $S_s$. We do this in the same way that we did for sequence alignment: $ALIGN(Ø,Ø) = 0$, $ALIGN((s_i,s_j),Ø)$ is the sum of the scores for gapping each base or base-pair in $(s_i,s_j)$, and likewise for $ALIGN(Ø,(t_k,t_l))$.

To compute an optimal global alignment, we find $ALIGN((s_1,s_m),(t_1,t_n))$ using these recurrences. We can compute a local alignment in exactly the same way as we do for a local sequence alignment; we prevent any suffix-suffix alignment from having a score below zero, and we start our traceback at the highest-scoring suffix-suffix alignment. We don't have to modify the pair-suffix alignments because pair-suffixes are bookended by aligned base pairs and hence don't have 'start gaps' or 'end gaps'.

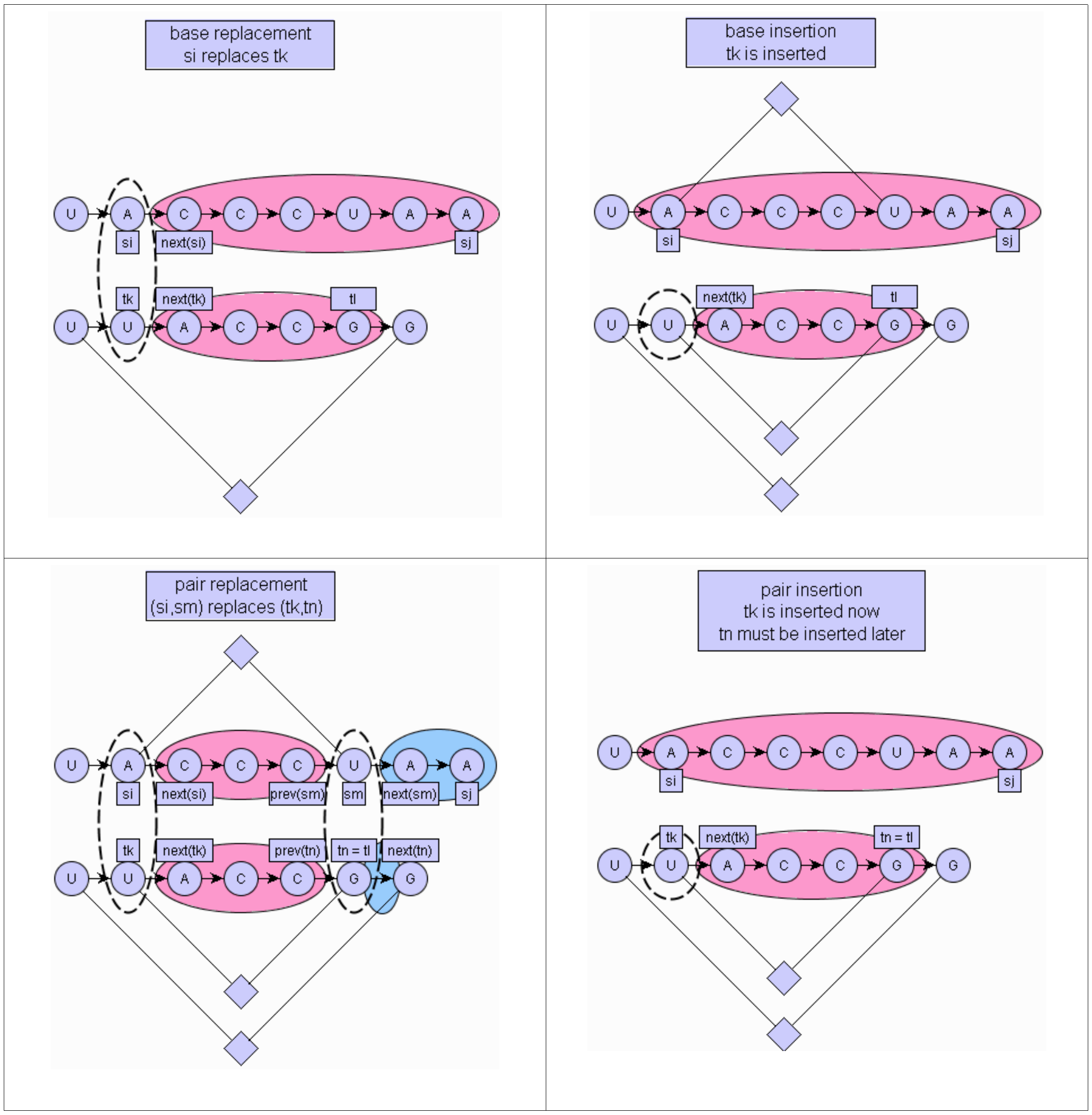


*Table 10: arc-annotated string editing operations during alignment. All images depict alignments of (si,sj) with (tk,tl), but some have different structures. Aligned bases are enclosed in dashes. Subalignments are colored pink or blue. Top left: base replacement. Top right: base insertion. Bottom left: pair replacement. Note that pair replacement splits into two subalignments - pink pair-suffixes 'inside' the paired bases and blue suffixes 'outside'. Since tn = tl, (next(tn),tl) is empty, so the second subalignment will have to insert (next(sm),sj). Bottom right: pair insertion. We only actually insert tk, paying half of the total cost for inserting the pair (tk,tn). All subalignments of (next(tk),tn) will have to insert tn, paying the other half of the cost. We guarantee that we'll insert tn in those subalignments; tn's pair, tk, is incompatible with them.*

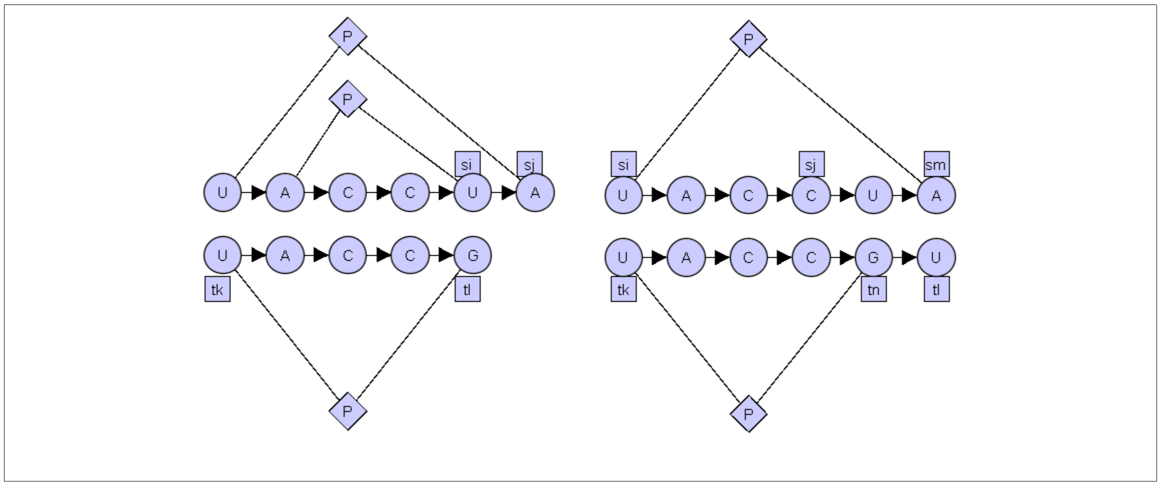


*Table 11: two ways to have incompatible base pairs during alignment. Left: si's partner appears before the substring si,sj. This occurs after pair indels, as in the previous table. Right: si's partner appears after si,sj. This can only occur if there are crossing base pairs, i.e. tertiary structure.*

Note that ($next(s_i)$,$prev(s_m)$) and ($next(t_k)$,$prev(t_n)$) are both pair-suffixes. All of the other suffixes considered by *ALIGN* are always of the same type as ($s_i$,$s_j$) and ($t_k$,$t_l$). Since we start with an alignment of two suffixes ($s_1$,$s_m$) and ($t_1$,$t_n$), *ALIGN* only has to compute alignments of suffixes to suffixes and pair-suffixes to pair-suffixes. Each pair in $S_p$ has O($m$) pair-suffixes, so there are O($mm_p$) pair-suffixes in $S$. Hence the alignment space complexity is O($mm_pnn_p$). To compute each subalignment, we do only a few constant-time lookups, so the time complexity is also O($mm_pnn_p$).

Earlier, I noted that incompatible base pairs could arise from tertiary structure. If $S$ and/or $T$ have tertiary structure, the algorithm will find the optimal alignment of $S$ and $T$ whose arcs form a valid secondary structure. For example, consider the crossing base pairs ($s_i$,$s_j$) and ($s_k$,$s_l$) such that $i < k < j < l$. If the algorithm chooses to replace either pair, then it splits the alignment into two parts that are incompatible with the other pair. Therefore the aligned base pairs in the optimal arc-annotated alignment are all non-crossing; the algorithm chooses the set of non-crossing base pairs that give the highest alignment score.

(Wang and Zhang, 2001) outline a two-pass method for aligning tertiary structures that takes advantage of this property. In the first pass, they run the algorithm as usual to find the optimal alignment of a secondary structure of $S$ with a secondary structure of $T$. In the second pass they run the algorithm again, but with two modifications. First, they remove all base pairs ($s_i$,$s_m$) and ($t_k$,$t_n$) that were aligned in the first pass. This way, the second pass can include base pairs that cross them. The second modification is to constrain the alignment so that $s_i$ must be aligned with $t_k$ and $s_m$ must be aligned with $t_n$. Thus the second pass aligns tertiary structure, but is constrained to respect the base pair matchings made by the first pass.[12] This procedure might not produce an optimal alignment of tertiary structure; the constraints from the optimal first pass might prohibit a good second pass alignment. However, this method appears to produce good alignments in practice. Most RNA structures have only a few tertiary interactions, so the optimal first pass does most of the work.

A final note: arc-annotated alignments can be thought of as matching “sequence *and* structure”, not “sequence *or* structure”. For example, if we align a structured RNA with its unstructured sequence, the arc-constraint would force us to gap every paired base. This is partially a theoretical point, because in practice we want structure-to-sequence alignments to do things that a structure-to-structure alignment might not, e.g. favor the alignment of paired bases in the structure with complementary bases in the sequence. However, it shows the restrictiveness of the arc-annotated representation - because the alignment imposes a consensus structure on all aligned RNAs, the alignment don't distinguish between structural and non-structural differences. This is an issue that I will address in this thesis.

12 (Wang and Zhang, 2001) do not explain their method in detail, but it appears to work in this fashion.

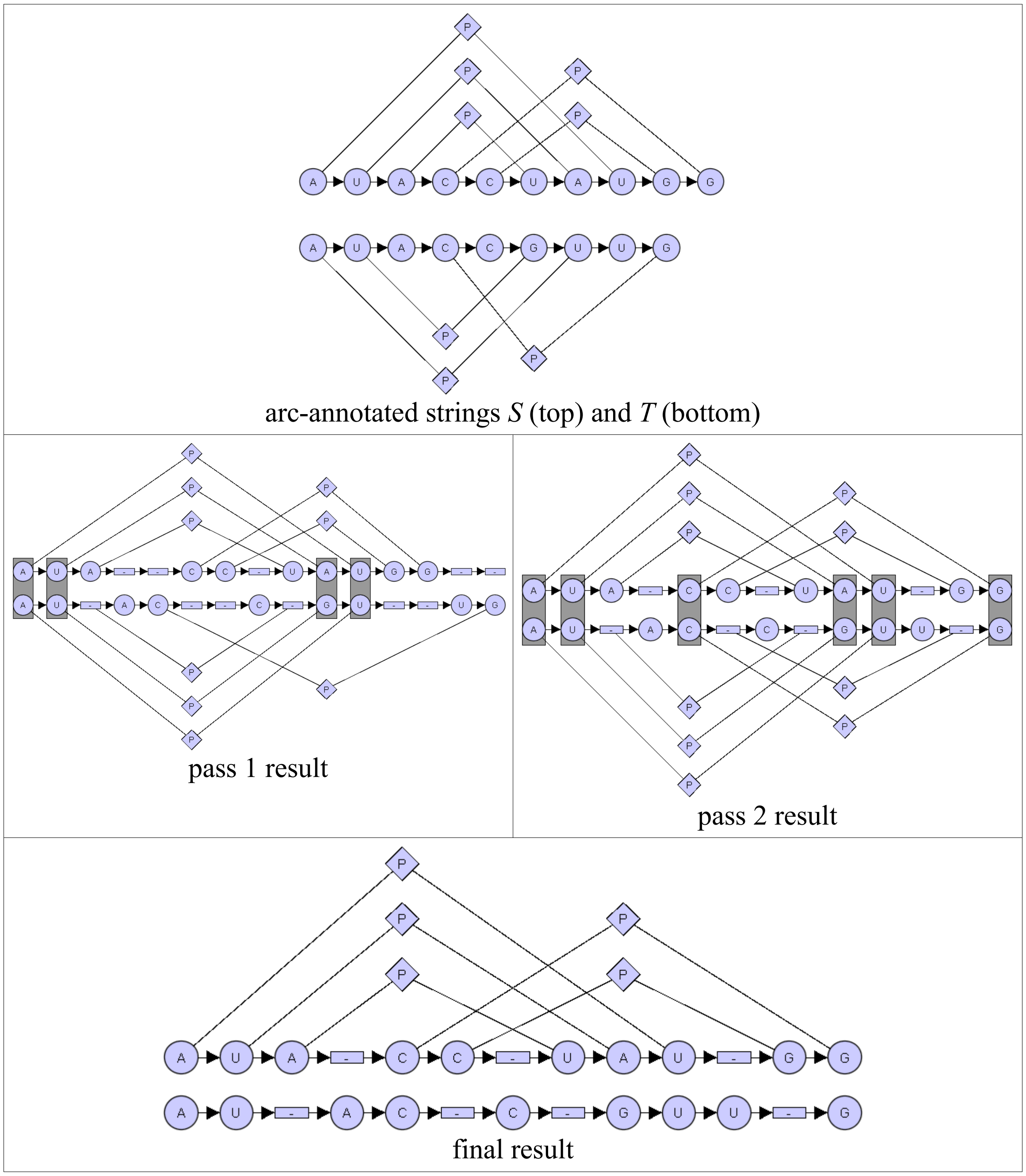


*Table 12: arc-annotated alignment of tertiary structures. Top: structures S (top) and T (bottom). Middle left: the first pass computes the optimal alignment of secondary structures of S and T. Constrained bases are highlighted in gray. Middle right: the second pass computes optimal alignment of remaining structure, ignoring the base-pairs for constrained bases. Bottom: the resulting arc-annotated alignment.*

### 2.5.2 Pairwise alignment of forests

RNA secondary structures can be compared and aligned as ordered trees or ordered forests. An early example is due to (Shapiro and Zhang, 1990), who used trees to represent RNA structure topology. Each node in a tree represents a type of loop. The root of a tree is the zero loop, and each internal node is an internal, bulge, or multibranch loop. All leaf nodes are hairpin loops. This is a very compact but coarse-grained representation; it tells us about an RNA's overall shape, but not about its individual base pairs.

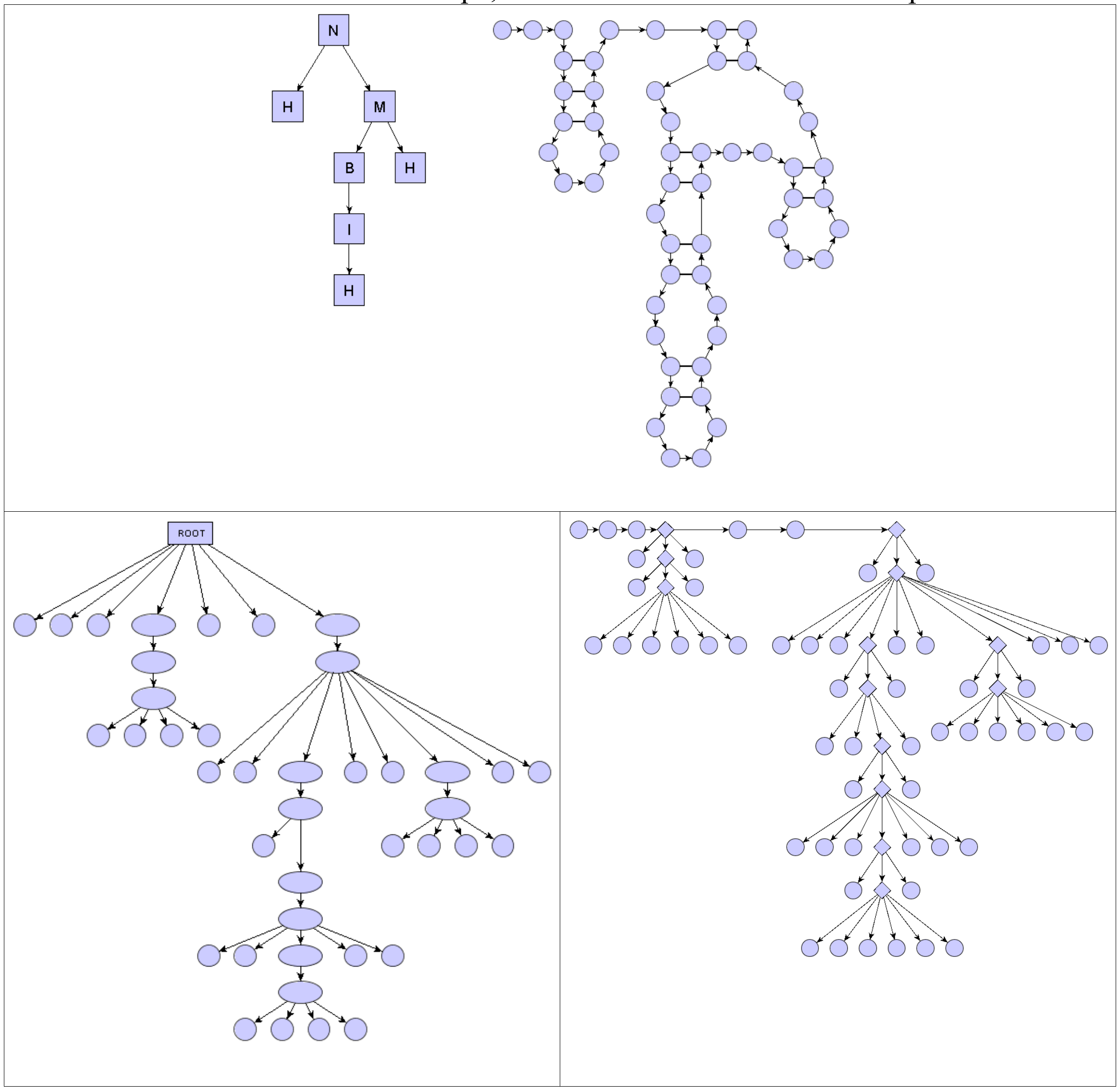


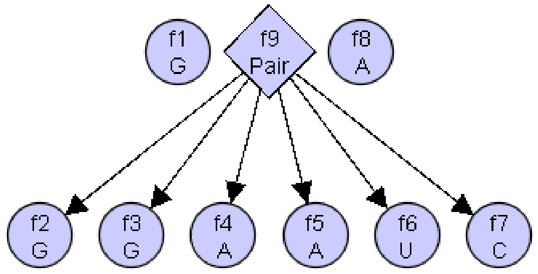
*Table 13: Schematics of several tree and forest representations. Top: coarse tree representation (left) of an RNA secondary structure (right). Bottom left: tree representation of (Hofacker et al., 1994). Pair nodes (ellipses) represent two nodes and the pair between them. Bottom right: forest representation of (Hochsmann, 2003). Pair nodes (diamonds) represent a pair between their leftmost and rightmost children.*

There are several more fine-grained tree representations for RNA structures. (Hofacker *et al.*, 1994), among others, places unpaired bases on tree leaves and base pairs on internal nodes. Hence each internal node represents two bases and the pair between them.[13] This is analogous to an arc-annotated string, in that each base pair ($s_i$,$s_j$) must be aligned as a whole unit; $s_i$ and $s_j$ cannot be aligned individually. (Hochsmann *et al.*, 2003), represents a base pair with a special 'pair' node and its paired bases as its leftmost and rightmost children. Unlike the other types of trees, this allows us to generalize sequence alignment. If a 'G' and a 'U' are paired in one structure and unpaired in the other, the nucleotides will still be matched; only the pair node will be gapped. This is the representation I will use in this thesis.

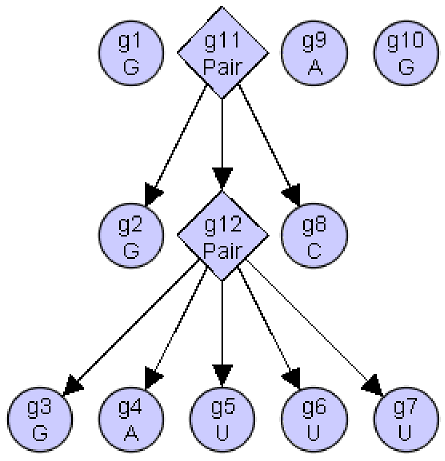
We'll define an alignment *A* of labeled, ordered forests *F* and *G* as follows. *A* is a forest in which each node has a pair of labels: one from a node in *F* and one from a node in G, or else one from either *F* or *G* opposite a gap. *A* must be a “componentwise projection” of both *F* and *G* (Hochsmann *et al.*, 2003). That is, we can recover *F* from *A* by erasing the the part of each node label in *A* that comes from *G*, deleting each base nodes that has a 'gap' label, and merging each pair node that has a 'gap' label with its parent node. It is useful to think about forest alignments as generalizations of sequence alignments. A sequence alignment is a componentwise projection of each of its sequences. Each node in *A* is like a column in a sequence alignment; the only difference is that each column of *A* is placed on a graph node rather than in a sequence. We score forest alignments in the same way as sequence alignments: summing the match, mismatch, and gap scores from each column. Hence sequence alignments are just forest alignments that have no pair nodes.

13 (Hofacker *et al.*, 1994) also use 'homeomorphically irreducible trees' which collapse consecutive internal nodes into single nodes representing stems, and consecutive leaf nodes into single nodes representing loops.

Forest *F*



Forest *G*



Alignment *A* of *F* and *G*

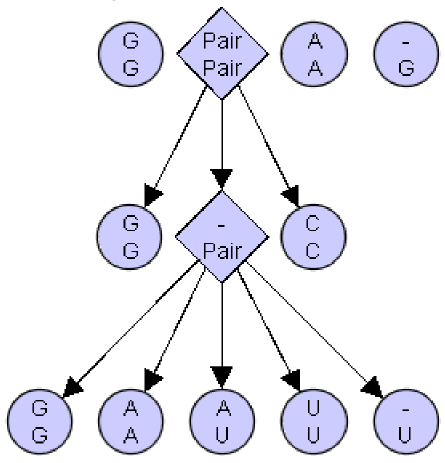


*Table 14: alignment of two RNA forests. Nodes in the forests are labeled with their nucleotide, and a pair of indexes (structure, sequence position). For example, $g_0$ is the first nucleotide in structure G. All pair nodes have 'sequence position' off the end of the sequence, e.g. $f_9$ when F only has sequence positions 1 through 8. Each node in the alignment represents a gap or the alignment of a node in F with a node in G.*

We'll need some definitions for describing these forests. For a forest $F$ with $|F|$ nodes $\{f_1,f_2,...,f_m\}$:

*Bases(F)* is the number of base nodes in $F$ (i.e. number of nodes with no children)
*Pairs(F)* is the number of pair nodes in $F$.
*MaxLoop(F)* is the maximum number of children of any node of $F$.[14]

14 For these forests, *MaxLoop(F)* is just the degree of *F*. I use this notation because the degree of partial-order graphs has a different meaning than *MaxLoop*, so it is useful to keep them distinct.

*next*($f_i$) is the right brother of $f_i$. If $f_i$ has no right brother, *next*($f_i$) = Ø
*pair*($f_i$) = ($f_k$,$f_l$), where ($f_k$) is the leftmost child of $f_i$ and $f_l$ is the rightmost child of $f_i$. If $f_i$ is not a pair node, *pair*($f_i$) = Ø.

($f_i$,$f_j$), where $f_j$ is a brother of $f_i$ that can be reached by repeatedly applying *next*, is a *closed subforest*. A closed subforest is to a forest as a subsequence is to a sequence.
($f_i$,$f_j$) is a *suffix* if *next*($f_j$) = Ø.
($f_i$,$f_j$) is a *nonsuffix* if *next*($f_j$) is not Ø.
I will typically use ($f_i$,$f_j$) to refer only to the sequence of brother nodes from $f_i$ to $f_j$; to indicate all of the nodes in that subforest, I will say *subforest*($f_i$,$f_j$).

*score*($f_i$,$t_k$) is the score for putting $f_i$ and $t_k$ at the same position in an alignment.

For example, in *F* in the table above, *Bases(F)* = 8, *Pairs(F)* = 1, and *MaxLoop(F)* = 6. *next*($f_2$) = $f_3$, and *pair*($f_9$) = ($f_2$,$f_7$). ($f_1$,$f_9$) is a nonsuffix, while ($f_9$,$f_8$) and ($f_2$,$f_6$) are suffixes. ($f_3$,$f_8$) is an invalid closed subforest because $f_7$ isn't a brother of $f_3$. ($f_5$,$f_2$) is an invalid closed subforest because $f_2$ can't be reached from $f_5$ by using *next*. Note that ($f_1$,$f_8$) has only three nodes in it - {$f_1$,$f_9$$f_8$} - while *subforest*($f_1$,$f_8$) contains all of the nodes in *F*.

Here I will present the alignment algorithm of (Jiang *et al.*, 1995). The original algorithm aligns trees and requires distance scores; I will describe a straightforward extension that aligns forests and uses similarity scores.[15]
I will explain the algorithm in terms of closed subforests, a term introduced by (Hochsmann *et al.*, 2003). Just as we can solve alignment of sequences by aligning their subsequences, we will solve alignment of forests by aligning their closed subforests. A global alignment of *F* and *G* is an alignment of the closed subforests ($f_1$,$f_8$) and ($g_1$,$g_{10}$) - that is, an alignment of all of *F* with all of G (as shown in the figure above). A local alignment of *F* and *G* is an alignment of the the closed subforests ($f_i$,$f_j$) and (gk,gl) that have the highest similarity.
As in sequence alignment, *ALIGN*(($f_i$,$f_j$),($g_k$,$g_l$)) maximizes over subalignments produced by replacement, insertion and deletion.

*ALIGN*(($f_i$,$f_j$),($g_k$,$g_l$)) = max of
*ALIGN*(($f_i$,$f_j$),($g_k$,$g_l$)) , replacement
*ALIGN*(($f_i$,$f_j$),($g_k$,$g_l$)) , insertion
*ALIGN*(($f_i$,$f_j$),($g_k$,$g_l$)) , deletion

This is a bit trickier than sequence or arc-annotated string alignment. The first tricky bit is that we have to consider aligning both *next*($f_i$) and *pair*($f_i$); in arc-annotated

15 see e.g. (Bille, 2003) for a variant that aligns forests, and see (Wang and Zhao, 2003), who base their parametric, affine-gapped, space-saving algorithm on a variant that uses similarity scores.

string alignment, we only needed to consider one *or* the other. In addition, we will have to consider both suffixes and nonsuffixes. First, let us consider a replacement operation:

```
$ALIGN((f_i,f_j),(g_k,g_l))$, replacement =
        $score(f_i,g_k)$
        $+ ALIGN(pair(f_i),pair(g_k))$
        $+ ALIGN((next(f_i),f_j), (next(g_k),g_l))$
```

For replacement, we first align $f_i$ and $g_k$ at the same position. This splits $(f_i,f_j)$ into two closed subforests. One of them is a loop defined by the paired bases $pair(f_i)$. If $f_i$ is a base node, then it doesn't have any paired bases and $pair(f_i) = Ø$. The second is $(next(f_i),f_j)$; this is the remainder of $(f_i,f_j)$ after $f_i$ and $f_i$'s children have been aligned. Likewise, we split $(g_k,g_l)$ into two closed subforests. The optimal alignment of $(f_i,f_j)$ is built from an optimal alignment of the first subforest of $(f_i,f_j)$ with the first of $(g_k,g_l)$ and an optimal alignment of the second subforest of $(f_i,f_j)$ with the second of $(g_k,g_l)$. This is just like the 'compatible pair' step of an arc-annotated alignment, in which we split an alignment into two subalignments. However, these forests represents base pairs with special 'pair nodes'. Note that it would be meaningless to align a nucleotide to a molecular bond, so we prohibit alignments between pair nodes and base nodes (e.g. we set ($score$(pairnode,basenode) = $-\infty$).

What happens for insertion or deletion? If we place a gap opposite gk, then we'll again split $(g_k, g_l)$ into two separate subforests: $pair(g_k)$ and $(next(g_k),g_l)$. Clearly, we will have to split $(f_i,f_j)$ into two subforests as well. We don't know which split of $(f_i,f_j)$ is best, so we'll have to try all possible splits. Each node $f_{split}$ in $(f_i,f_j)$ splits it into two subforests: one starting at $f_i$ and ending at $f_{split}$, and another starting after $f_{split}$ and ending at $f_j$. Also, although the notation below doesn't show it, we must try splits in which we align the entire $(f_i,f_j)$ with either $pair(g_k)$ or $(next(g_k),g_l)$ and align Ø with the remaining subforest of $G$. For example, if $g_k$ is a base node then $pair(g_k) = Ø$, so we don't want to align any subforest of $F$ with $pair(g_k)$.

Note that even if $(f_i,f_j)$ is a suffix, $(f_i,f_{split})$ will be a nonsuffix (unless $f_j = f_{split}$). Therefore, unlike arc-annotated string alignment, we have to consider nonsuffixes.

```
$ALIGN((f_i,f_j),(g_k,g_l))$, insertion =
        $score(-,g_k)$
        + max over $f_{split}$ in $(f_i,f_j)$ {
                $ALIGN((f_i,f_{split}),pair(g_k))$
                $+ ALIGN((next(f_{split}),f_j), (next(g_k),g_l))$
        }
```

Deletion is just a mirror image of insertion - we place a gap opposite $f_i$, then maximize over all splits $g_{split}$ in $(g_k,g_l)$. Finally, we have to initialize this algorithm just as we did for the others:

$ALIGN(\emptyset,\emptyset) = 0$
$ALIGN((f_i,f_j),\emptyset) = \text{sum } \{f_a \text{ in } subforest(f_i,f_j)\ \} \text{ score}(f_a,-)$
$ALIGN(\emptyset,(g_k,g_l)) = \text{sum } \{g_b \text{ in } subforest(g_k,g_l)\ \} \text{ score}(-,g_b)$

What is the complexity of a dynamic programming algorithm based on these recurrences? Since each pair defines a loop, $Pairs(F)MaxLoop(F) = O(|F|)$, and there are $O(MaxLoop(F)^2)$ closed subforests defined on each loop.[16] So there are $O(Pairs(F)MaxLoop(F)^2) = O(|F|MaxLoop(F))$ closed subforests in F. To store alignments of each subforest of *F* with each subforest of G, we would need $O(|F||G|\ MaxLoop(F)MaxLoop(G))$ space. For each subforest-subforest alignment we have to check $O(MaxLoop(F))$ split points for insertions and $O(MaxLoop(G))$ split points for deletions. Hence $O(|F||G|MaxLoop(F)MaxLoop(G)(MaxLoop(F) + MaxLoop(G))$ is an obvious time complexity bound.

However, if we look at the algorithm more carefully, we can see that its complexity is a bit better. Do we really have to align all closed subforests? After all, to align two sequences, we only align suffixes, not all subsequences. Let us try a similar approach here.

When will we need to consider nonsuffixes? When we indel a pair node, we have to divide our alignment into two subalignments. If $(f_i,f_j)$ was a nonsuffix and $(g_k,g_l)$ was also a nonsuffix, then we would have to consider nonsuffix-nonsuffix subalignments - the subalignment $ALIGN((next(f_i),f_j), (g_{split},g_l))$ would be between two nonsuffixes. However, as long as either $(f_i,f_j)$ or $(g_k,g_l)$ is a suffix, this division won't require us to consider a nonsuffix-nonsuffix subalignment (as shown in the next figure). So we only need nonsuffix-nonsuffix subalignments to compute the values of other nonsuffix-nonsuffix subalignments. Since we start with an alignment of two suffixes - the alignment of the whole of *F* with the whole of *G* - we never have to compute any nonsuffix-nonsuffix subalignments.

16 Just as there are $O(n^2)$ substrings of a string of length n

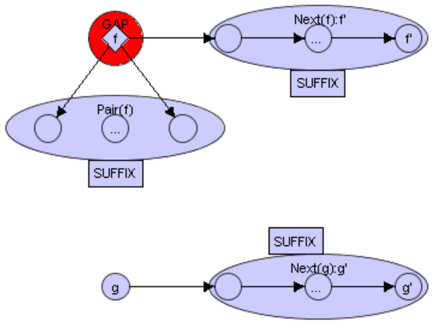
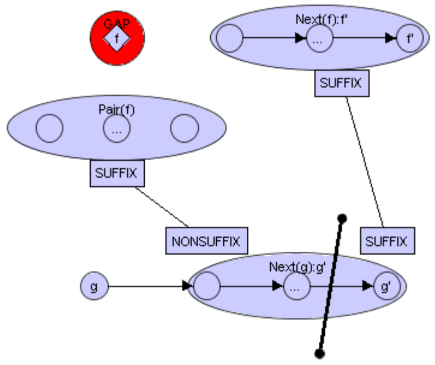
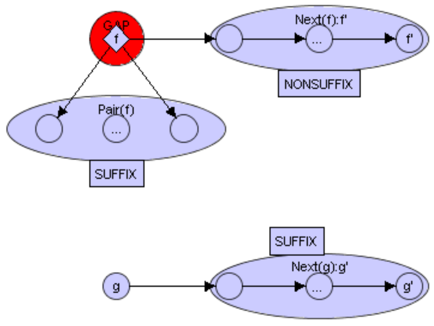
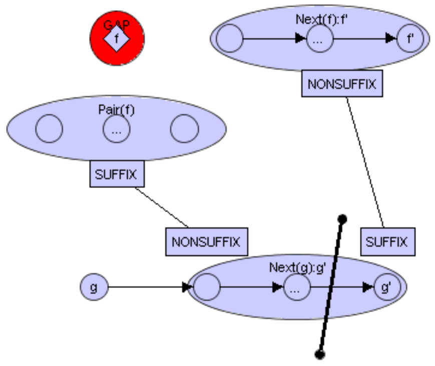
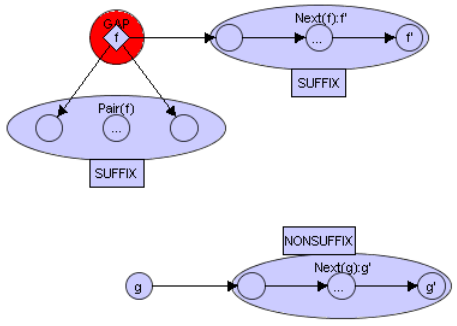
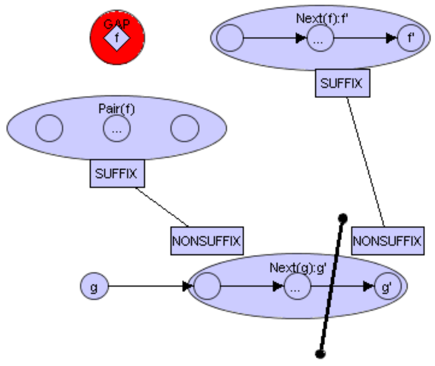
| | |
|---|---|
| A gap in a suffix-suffix alignment | produces suffix-nonsuffix, suffix-suffix |
|   GAP f Next(f):f' ... f' SUFFIX Pair(f) ... SUFFIX SUFFIX g Next(g):g' ... g' |   GAP f Next(f):f' ... f' SUFFIX Pair(f) ... SUFFIX NONSUFFIX SUFFIX g Next(g):g' ... g' |
| A gap in a nonsuffix-suffix alignment | produces suffix-nonsuffix, nonsuffix-suffix |
|   GAP f Next(f):f' ... f' NONSUFFIX Pair(f) ... SUFFIX SUFFIX g Next(g):g' ... g' |   GAP f Next(f):f' ... f' NONSUFFIX Pair(f) ... SUFFIX NONSUFFIX SUFFIX g Next(g):g' ... g' |
| A gap in a suffix-nonsuffix alignment | produces suffix-nonsuffix, suffix-nonsuffix |
|   GAP f Next(f):f' ... f' SUFFIX Pair(f) ... SUFFIX NONSUFFIX g Next(g):g' ... g' |   GAP f Next(f):f' ... f' SUFFIX Pair(f) ... SUFFIX NONSUFFIX NONSUFFIX g Next(g):g' ... g' |

*Table 15: So long as we don't start with a nonsuffix-nonsuffix alignment, we will never have to consider one. This is because the all other types of alignment (Suffix-suffix, nonsuffix-suffix, and suffix-nonsuffix) can be computed without considering nonsuffix-nonsuffix subalignments. 'Split' points are depicted as black slashes that divide the subforest of G into two adjacent subforests.*

How much time and space does this save? Well, there are $O(|F|)$ suffixes in F, since there is one suffix for each node. If we subtract the suffixes from the total number of closed subforests, we see that there are $O(|F|MaxLoop(F) - |F|) = O(|F|MaxLoop(F))$ nonsuffixes in F. Hence there are:

| | |
|---|---|
| 1. $O(\|F\|\|G\|)$ | suffix-suffix subalignments |
| 2. $O(\|F\|\|G\|MaxLoop(G))$ | suffix-nonsuffix subalignments |
| 3. $O(\|F\|MaxLoop(F)\|G\|)$ | nonsuffix-suffix subalignments |
| 4. $O(\|F\|MaxLoop(F)\|G\|MaxLoop(G))$ | nonsuffix-nonsuffix subalignments |

Since we only have to consider 1, 2, and 3, the space complexity is better than the naive approach: we only need to compute and store $O(|F||G|(MaxLoop(F)+MaxLoop(G)))$ subalignments. The time complexity is also improved: we check $O(MaxLoop(F)+MaxLoop(G))$ split nodes for each subalignment, so the total time is $O(|F||G|(MaxLoop(F)+MaxLoop(G))^2)$.

Incidentally, note that we only have to consider splits and nonsuffixes of *F* when we gap pair nodes in G. If *G* has no base pairs, then we don't have to compute any nonsuffix-suffix alignments, and we don't have to do $O(MaxLoop(F))$ split node tests per step. Then the algorithm requires only $O(|F||G|MaxLoop(F))$ space and $O(|F||G|MaxLoop(F)^2)$ time. If neither *F* nor *G* has base pairs, then the algorithm only needs $O(|F||G|)$ space and time - it reduces to Needleman-Wunsch sequence alignment. So with these modifications, forest alignment is a generalization of classic sequence alignment.[17]

Can we modify this algorithm for local alignment, in which we want to find the closed subforests $(f_i,f_j)$ and $(g_k,g_l)$ with the highest similarity? Both $(f_i,f_j)$ and $(g_k,g_l)$ could be nonsuffixes. A simple method used by (Hochsmann *et al.*, 2003) is to simply bite the bullet and compute all nonsuffix-nonsuffix alignments. This is the naive algorithm that runs in $O(|F||G|MaxLoop(F)MaxLoop(G)(MaxLoop(F)+MaxLoop(G)))$ time and $O(|F||G|MaxLoop(F)MaxLoop(G))$ space. It would be surprising if this was the best we could do. After all, we don't have to compute all subsequence-subsequence alignments for local sequence alignment, so why should we have to compute all subforest-subforest alignments for local forest alignment?

(Jansson *et al.*, 2004) give a two-pass method to locally align two forests without increasing the time and space complexity. In the first pass we compute a global alignment as usual, storing its results in a matrix $M_{\text{global}}$. In the second pass we recompute the suffix-suffix alignments and store them in a matrix $M_{\text{local}}$. However, the second pass computes its alignments with a few modifications reminiscent of the 'most similar subsequence' alignment algorithm (discussed in section 2.2). The first modification is to set the maximum cost of a suffix-suffix alignment to 0. The second modification is that we start our alignment traceback at the highest-scoring subalignment in $M_{\text{local}}$, rather than only starting the traceback at the 'entire forest' suffixes like we did in the global case. These

17 We could also get the algorithm to do $O(|F||G|)$ sequence alignment by transforming each sequence *S* so that $next(s_n) = \emptyset$ and $pair(s_n) = (s_{n+1},s_{n+1})$.

two modifications are just like the ones for 'most similar subsequence' alignment - they let us put any number of 'cost-free' indels at the start or end of a suffix.

The crucial third modification is as follows. Whenever we look up the value of a subalignment with $pair(f_i)$ or $pair(g_k)$, we look up that subalignment's score in $M_{\text{global}}$ instead of $M_{\text{local}}$. That is, we don't allow any 'free' gaps in subforests within our alignment. We modify the recurrences in this manner:

```
$ALIGN_{local}((f_i,f_j),(g_k,g_l))$, replacement =
    $score(f_i,g_k)$
    + $ALIGN_{global}(pair(f_i),pair(g_k))$
    + $ALIGN_{local}((next(f_i),f_j), (next(g_k),g_l))$

$ALIGN_{local}((f_i,f_j),(g_k,g_l))$, insertion =
    $score(-,g_k)$
    + max over $f_{split}$ in $(f_i,f_j)$ {
        $ALIGN_{global}((f_i,f_{split}),pair(g_k))$
        + $ALIGN_{local}((next(f_{split}),f_j), (next(g_k),g_l))$
    }
```

Now the highest scoring alignment can be between any two subforests $(f_i,f_j)$ and $(g_k,g_l)$. The alignment score will include gap costs for all nodes in $subforest(f_i,f_j)$ and $subforest(g_k,g_l)$, but we get free 'start gaps' and 'end gaps' for all other nodes.

As I implied earlier in this section, there are several interesting modifications to this algorithm which could also be used for other algorithms that I develop in this paper. In particular, (Wang and Zhao, 2003) modify forest alignment to use affine gap penalties and to reduce its space complexity. They implement affine gap penalties in a way analogous to traditional affine-gapped sequence alignment (Gotoh, 1982). Reducing the space complexity is a little trickier. The idea is as follows. For each subalignment A, there is a set of other subalignments $A_{1\ldots n}$ that maximize over A. What if all $A_{1\ldots n}$ have been computed and none of them chose A as an optimal subalignment? Then we know that the optimal alignment doesn't include A, so we don't have to store its score anymore. (Wang and Zhao, 2003) show that if we delete these unused subalignments, the space complexity for global alignment is only $O(MaxLoop(F)\log(|F|)|G|(MaxLoop(F)+MaxLoop(G)))$.

## *2.6 Alignment of multiple RNA secondary structures*

Despite the large number of algorithms for pairwise comparison of RNA secondary structures, only a few methods have been proposed for multiply aligning them. All of them are progressive alignment approaches. We can divide them into two categories: sequence based methods and structure-based methods.

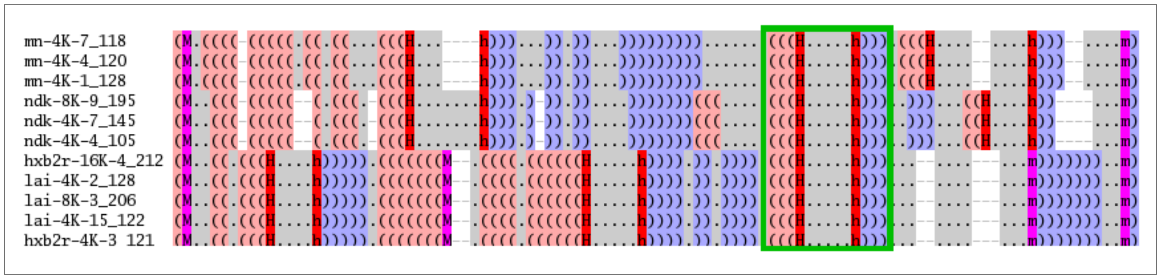
*Sequence-based methods* represent RNA structures as text strings and align them with sequence alignment algorithms. The challenge is to use a clever scoring scheme to get sequence alignment algorithms to implicitly take structure into account. However, although the structural information encoded in a text string can guide a sequence alignment, it cannot constrain the alignment to be structurally valid. For example, even if the 5' sides of two stems are aligned with each other, the 3' sides may not be.

*Structure-based methods* represent RNAs using data structures that explicitly represent base pairing interactions. Some of them represent base pairs as constraints on an alignment of sequences, whereas others represent base pairs as entities which can be deleted and inserted individually. In either case, these methods can enforce structural constraints explicitly and thus guarantee valid alignments.

### 2.6.1 Sequence-based methods

The first approach to multiple alignment of RNA structures is due to (Shaprio, 1988), who represented RNA structures as text strings and aligned them with a standard progressive multiple sequence alignment algorithm. These text strings only had characters to represent topological features (e.g. “B” for a bulge loop, “M” for a multibranch loop, etc.), but strings can be enriched with individual base-pairs (Hofacker *et al.*, 1994) and sequence information using variations on 'bracket' notation.[18] The advantage of this approach is that it is as fast as multiple sequence alignment - e.g. about $O(n^2k^2)$ time for tree-guided progressive alignment of $k$ structures of at most $n$ bases each. However, alignments often include structurally invalid regions. In (Bromberg-Martin *et al.*, unpublished), I use an alignment method of this type as part of a system for exploring large datasets of secondary structures. The alignments can be used to quickly cluster and visualize many structures at once, but they are not suitable for rigorous genetic analysis.

18 'bracket' notation is explained in section 2.4



```
mn-4K-7_118       (M.((((-(((((.((.((...(((H...----h)))...)).))...)))))))))......(((H.....h))).(((H....--....h)))--....m)
mn-4K-4_120       (M.((((-(((((.((.((...(((H...----h)))...)).))...)))))))))......(((H.....h))).(((H....--....h)))--....m)
mn-4K-1_128       (M.((((-(((((.((.((...(((H...----h)))...)).))...)))))))))......(((H.....h))).(((H....--....h)))--....m)
ndk-8K-9_195      (M..(((-(((((--(.(((.-(((H.......h))).)-)).))....)))))))(((....(((H.....h)))..)))...((H....h))----...m)
ndk-4K-7_145      (M..(((-(((((--(.(((.-(((H.......h))).)-)).))....)))))))(((....(((H.....h)))..)))...((H....h))----...m)
ndk-4K-4_105      (M..(((-(((((--(.(((.-(((H.......h))).)-)).))....)))))))(((....(((H.....h)))..)))...((H....h))----...m)
hxb2r-16K-4_212   (M..((.(((H....h))))).(((((((M--.((((.((((((H.....h)))).)).))))(((H.....h)))..--....---...m)))))))..m)
lai-4K-2_128      (M..((.(((H....h))))).(((((((M--.((((.((((((H.....h)))).)).))))(((H.....h)))..--....---...m)))))))..m)
lai-8K-3_206      (M..((.(((H....h))))).(((((((M--.((((.((((((H.....h)))).)).))))(((H.....h)))..--....---...m)))))))..m)
lai-4K-15_122     (M..((.(((H....h))))).(((((((M--.((((.((((((H.....h)))).)).))))(((H.....h)))..--....---...m)))))))..m)
hxb2r-4K-3 121    (M..((.(((H....h))))).(((((((M--.((((.((((((H.....h)))).)).))))(((H.....h)))..--....---...m)))))))..m)
```

*Table 16: portion of a multiple string alignment using secondary structure and topology information (Bromberg-Martin et al., unpublished). "H" and "h" indicate the start and end of hairpin loops, and "M"/"m" are for multibranch loops. All of the structures have a multibranch loop in this region, but only one of its hairpin loops is conserved (green box). Note that structurally dissimilar regions can match each other, and 'topology' characters can even align with nucleotides (e.g. to the right of the box).*

The `MARNA` program of (Siebert and Backofen, 2003) multiply aligns RNA structures by making clever use of the `T-COFFEE` sequence alignment algorithm (Notredame *et al.*, 2000). `T-COFFEE` uses a library of pairwise sequence alignments to constrain a progressive multiple sequence alignment. The main idea is that bases matched in pairwise alignments should still be matched in a multiple alignment. The resulting constraints greatly improve multiple alignment accuracy. `MARNA` feeds T-COFFEE a library of RNA alignments made using the pairwise RNA tree editing method of (Jiang *et al.*, 2002). Thus the resulting mulitple alignment is constrained to take structural similarities into account. The drawbacks of `MARNA` are that it is relatively slow (`T-COFFEE` runs in O($n^2k^3$) time) and that, despite its alignment constraints, it still does not guarantee that the multiple alignment is structurally valid.

The `StructMiner` program of (Yang and Blanchette, 2004) guides RNA multiple sequence alignment using base-pairing probability matrices, a method first introduced for pairwise alignments by (Bonhoeffer *et al.*, 1993). A base-pairing probability matrix P is defined for a sequence $S$ such that $P_{ij}$ = Pr{ base $s_i$ is paired to base $s_j$ }. If P represents a known RNA secondary structure, then all of its entries are 0 or 1. However, the great advantage of this approach is that we are rarely certain of an RNA's true structure, but we can estimate an RNA's base-pairing probabilities using thermodynamic and/or phylogenetic methods (e.g. the popular program `mfold` (Zuker, 1999)). Thus this alignment method can be used in cases when purely structural approaches cannot.

To compute pairwise alignments, `StructMiner` uses a method very much like that of (Bonhoeffer *et al.*, 1993). The method involves standard pairwise alignment of structures represented in a manner very similar to 'bracket' notation, but instead of bases being categorized as one of {paired upstream, paired downstream, unpaired}, a base's match and mismatch scores depend on the probability that it is in each of those categories. For example, a base with a 50% probability of being unpaired will not match perfectly to a base with a 100% probability of being unpaired. To compute multiple alignments, `StructMiner` uses a progressive profile approach. It is somewhat unconventional in that it simply merges structures into the multiple alignment one by one without use of a guide tree. Once this is done, it scans through the alignment to find and fix structurally inconsistent regions which are within a small distance of their correct location. Finally, `StructMiner` rejects any remaining base-pairings which are structurally invalid (e.g. tertiary structure and bases paired to multiple partners).

Because `StructMiner` uses a simple progressive alignment without computing a guide tree, it is very fast - $O(kn^2)$ time and $O(n^2)$ space (although the base-pairing matrices must be computed by an $O(n^3)$ prediction algorithm (Hofacker *et al.*, 1994)). In addition, it appears to give quite good results for aligning small numbers of structurally related RNAs, even when they have low sequence similarity. I should note that (Hofacker *et al.*, 2004) (discussed below) give a more rigorous, structure-based method for multiple alignment of base-pairing probability matrices. (Yang and Blanchette, 2004) reference Hofacker *et al.*, but do not mention their work or compare the results of the two algorithms. It would be interesting to compare their results, especially since Hofacker *et al.* found that using the string alignment method, even if only to produce a guide tree for their more rigorous structural method, significantly reduced the quality of alignments.

## 2.6.2 Structure-based methods

(Wang and Zhang, 2004) represent structures as arc-annotated strings and their alignments as arc-annotated alignments of strings. They extend the algorithm of (Wang and Zhang, 2001) to approximate the optimal alignment of two arc-annotated alignments. Arc-annotations strongly constrain the alignment to be structurally valid, since all nucleotides aligned in a column have exactly the same structure. *ALIGN*(structure alignment, structure alignment) takes $O(n_1 n_2 p_1 p_2)$ time and $O(n_1 n_2 + n_1 k_2 + n_2 k_1)$ space, where $n$ is the number of columns (i.e. nucleotides), $k$ is the number of rows (i.e. structures), and $p$ is the number of pairs in each alignment. To the best of my knowledge, there is no publicly available implementation of this method, and (Wang and Zhang, 2004) do not show the results of any multiple alignments. Therefore it is difficult to evaluate the effectiveness of this method. However, some authors have argued that the structural constraints of this type of RNA comparison model are too strong, since two unpaired bases can never align to two paired bases. To edit a "U-G" pair into "G G", we must treat it as two insertions of single bases followed by two deletions of paired bases, but it seems more natural to treat it as a "U to G" substitution and a deletion of a base-pair (Jiang *et al.* 2002). In essence, arc-annotated alignments represent only a single consensus structure.

The `pmmulti` program of (Hofacker *et al.*, 2004) aligns RNA sequences whose structures are described by base-pairing probability matrices. As discussed above, this approach can be used for aligning RNAs with known structure, but is also useful when we can only predict an RNA's structure probabilistically. `pmmulti`'s pairwise algorithm is a variant of Sankoff's maximum circular matching algorithm for simultaneously aligning and folding two RNA sequences (Sankoff, 1985). Their multiple alignments use a standard progressive profile approach.

`pmmulti` appears to be considerably more accurate for predicting and aligning secondary structures than sequence-based methods such as `MARNA`. However, it has two major drawbacks. First, the Sankoff algorithm is very slow. Enforcing several restrictions on the size of indels reduces the time complexity from O($n^6$) to O($n^4$) and the space complexity from O($n^4$) to O($n^3$), but it is still only practical for rather small sequence lengths. This is an especially limiting factor when doing $n(n\text{-}1)/2$ pairwise alignments to create a guide tree for the final ($n$-1) sequence alignments. The authors mitigate this to some extent by including an option to use the fast but less accurate pairwise algorithm of (Bonhoeffer *et al.*, 1993) for constructing the guide tree, but this reduces the quality of alignments. Second, in a manner analogous to arc-annotated alignments, each of `pmmulti`'s alignments cannot represent fine structural variation; they represent only a single consensus probability matrix.

The `RNAForester` program of (Höchsmann *et al.* 2004) represents RNA structures as forests and their alignments as 'RNA profile' forests. RNA profiles are exactly analogous to row-column sequence alignment profiles. Each node of an RNA profile represents a 'column' of an alignment, and its label counts the number of nodes of each type in that column. The cost of matching two nodes depends on all of bases, pairs, and gaps they contain. See the following figure for an example of an alignment between two RNA structures and the resulting RNA profile.

`RNAForester` implements *ALIGN*(structure, structure) and *ALIGN*(RNA profile, RNA profile) with the pairwise alignment algorithm of (Jiang *et al.* 1995). Hence each alignment takes $O(|F||G|(MaxLoop(F) + MaxLoop(G))^2)$ time and $O(|F||G|(MaxLoop(F) + MaxLoop(G)))$ space. For most real RNA molecules, for which the maximum loop size is practically constant, `RNAForester` should be the fastest structure-based multiple alignment algorithm. Another advantage of this approach is that the forest representation doesn't impose the same set of arcs or base-pairing probabilities on each structure. Since base-pairs are just like any other node in a RNA profile, they can be matched, deleted, and inserted. Thus "U-G" can now align with "G G" with the intuitive editing cost of a base substitution and a base-pair deletion.

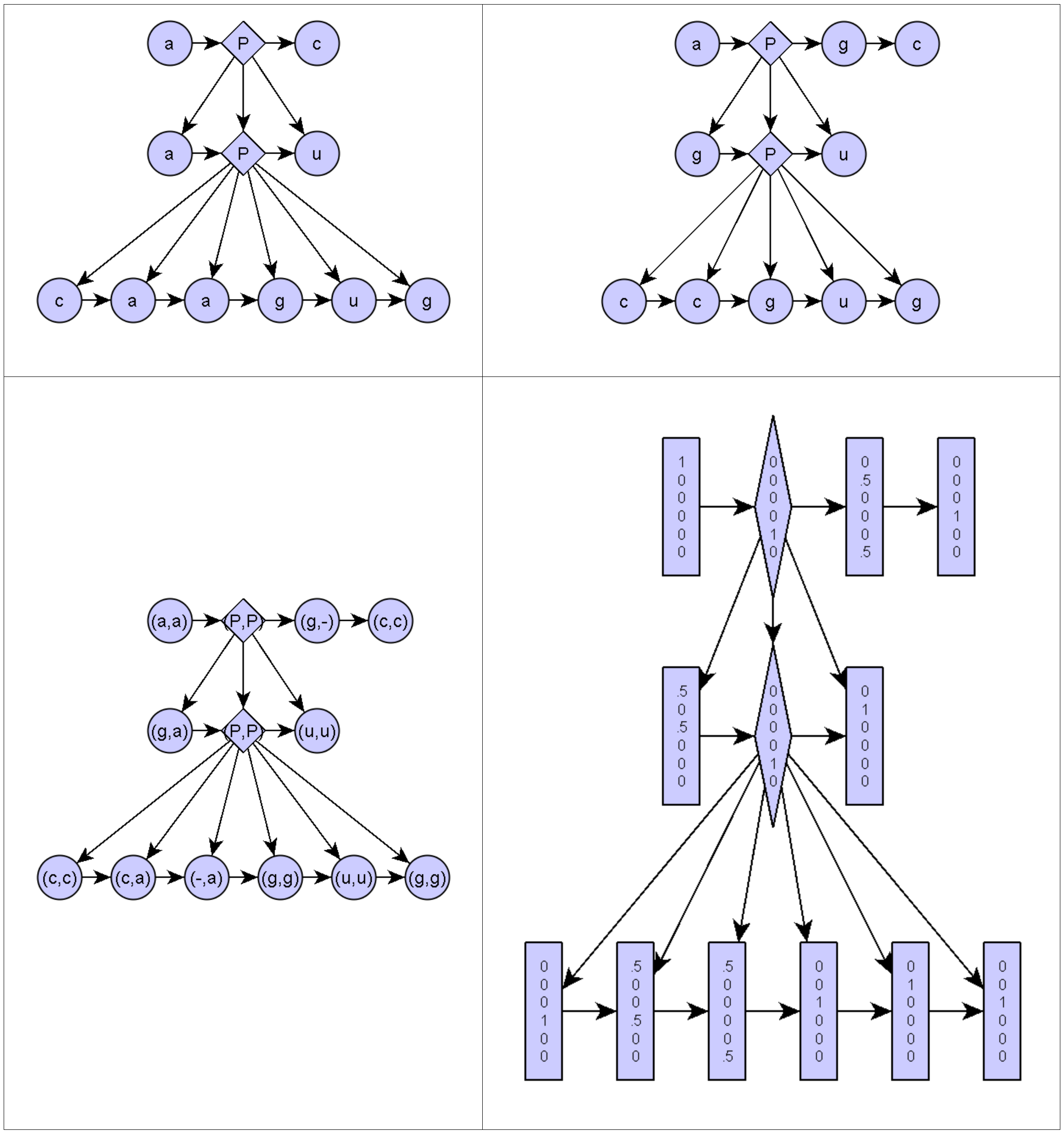


*Table 17: Alignment between two RNA structures of (Höchsmann et al., 2004). Top left: structure X. Top right: structure Y. Bottom left: Alignment between X and Y. Bottom right: RNA profile created from the alignment. Each node holds the fraction of aligned nodes that are in {a,u,g,c,pair,-}, respectively.*

`RSMatch` (Liu *et al.*, 2005) represents structures as trees and aligns them with a very fast algorithm - O(*mn*), the same complexity as sequence alignment. Their method is meant for aligning very large numbers of structures, e.g. for database searches. The method appears to be effective at detecting conserved motifs in sequences based on folds computed by structure prediction programs. However, it has a more restrictive editing model than any other structural alignment method mentioned in this paper. First, like (Hofacker *et al.*, 1994), `RSMatch` represents base pairs as whole, unbreakable entities. In addition, it represents sequences of unpaired bases as unbreakable 'circles' as well. We can achieve a similar effect with the forest alignment algorithm of (Jiang *et al.*, 1995) by ignoring all 'splits'; when $g_k$ is gapped, we gap its entire subforest as well. Then we never need to compute alignments with nonsuffixes and never need to consider split nodes, so the time and space complexity becomes $O(|F||G|)$, i.e. quadratic like `RSMatch`. With this sort of editing model, RNAs that have similar sequences but regions of dissimilar structure cannot be aligned. (Liu *et al.*, 2005) cast this editing model in a positive light, arguing that it produces structurally consistent alignments by preserving the integrity of paired and unpaired regions of aligned RNAs. As I will discuss in the next section, this approach is best suited to tasks such as pattern-matching and database search, but its multiple alignments do not capture the full range of structural variation between RNAs.

### 2.6.3 Room for improvement?

How can we improve multiple alignments of RNA secondary structures? This is a special case of the general question: what do we want out of any 'alignment'? The goal of any alignment is to 'line up' or overlay many individual data structures to show their similarities and variations. Within that framework, there are as many objectives for alignment as there are scoring functions, but there are some general principles we can apply. Here I will focus on two criteria for RNA structure alignments which I believe are reasonable and intuitive, but which existing methods do not fully address.

The first criterion is the naturalness of an alignment. An alignment should be able to handle arbitrary sets of RNAs, and should include a faithful representation of each individual RNA sequence and structure. A natural representation makes it possible to see the variations within a set of RNAs, not just a certain type of similarity. The second criterion is that the alignment should generalize both sequence and structure alignments. Structural features should be able to inform the alignment of nucleotides and vice versa. Our scoring function alone should determine whether we give first priority to sequence similarities or structural similarities; our alignments shouldn't have a built-in bias either way.

If we desire a natural representation for RNA structure alignments, we cannot rely on sequence-based methods. Despite various clever ways of embedding structural information in text strings, they use an inherently unnatural representation for RNA structure and hence are prone to structurally invalid alignments. It is possible to postprocess string alignments to enforce structural validity, but this adds structural considerations 'after the fact' rather than taking them into account directly during the alignment. `MARNA` and `StructMiner` take steps in the right direction. I am particularly intrigued by `StructMiner` because its reported performance seems to be much better than we would expect based on the results of (Hofacker *et al.*, 2004). Nevertheless, I will concentrate on structure-based methods in this thesis.

Structure-based methods are an improvement because they can stay faithful to the structures of individual RNAs. However, representing an *alignment* of RNA sequences and structures is still problematic.

Consider sequence alignments. Sequence alignments find the commonalities between sequences, but that is only one reason they are useful. They also tell us a great deal about the variation within a set of aligned sequences. Even a row-column profile tells us a great deal about each position's sequence variation. We can use this data to look for evolutionary or functional relationships, whether by quantitative analysis or by visualization. For example, sequence logos (Scheneider and Stephens, 1990) display subtle sequence variations in information-theoretic terms that can reveal functionally important nucleotides. Partial-order graphs contain each aligned sequence as a subgraph, and hence they preserve even more sequence variation than row-column profiles. Programs like `POAVIZ` (Grasso *et al.*, 2003) that visualize PO-MSAs can illustrate the complex evolutionary histories of large multi-domain proteins.

On the other hand, existing RNA structure alignment methods impose strong restrictions on structural variation. The arc-annotated alignments of (Wang and Zhang, 2004) represent only a single consensus secondary structure. Furthermore, they do not make a strong distinction between alterations in structure and alterations in sequence. The alignments are constrained to put structural similarity 'first', and make up for the difference with nucleotide indels. For example, if two RNAs have identical sequences but different structural features, then an arc-annotated alignment will always have to insert gaps in the sequences. In this sense, these alignments do not align sequence features or structure features, but rather "sequence *and* structure" features. `RSmatch` alignments also have this property due to their restrictive editing model.

`pmmulti`'s probabilistic alignments represent an ensemble of possible RNA structures. These alignments are also relatively flexible: they can match an unpaired 'A' with a paired 'A', depending on the scoring function. Unfortunately, its base pairing probability matrices represent consensus variations rather than variations between individual structures. In general, if $P_{ij} = 0.3$, it could be because all structures have $P_{ij} = 0.3$; alternately, one structure is certain to pair *i* with *j* but other structures are unlikely to do so. We can see this effect in a 'toy example' alignment of RNAs with known structure in (Hofacker *et al.*, 2004). A "G-U" base pair known to exist for an RNA with probability 1.0 actually gets deleted from the structure during the course of alignment. This happens because the other RNAs didn't have complementary nucleotides at those positions, and therefore had zero pairing probability at that position. In this sense, `pmmulti` boils down structural variations into a consensus probability matrix.

The RNA profiles of `RNAForester` are an improvement because they represent variations in both sequence and structure. For example, a profile pair node that has many gaps is an infrequent but possible base pair. However, RNA profiles can only represent a small subset of mappings between secondary structures. Each RNA is a componentwise projection of its RNA profile, and the profile itself is a secondary structure. Hence all RNAs must be 'nested' within a single secondary structure scaffold, and each RNA added to the profile further constrains the scaffold. A few mutations may make a small alteration to an RNA's structure that prevents it from being nested within the same scaffold. In that case, forest alignments will create artifactual indels to force the two molecules into a single secondary structure. For example, if any structure contains '`(...)`', then the central nucleotide in the loop can never be paired outside of that loop.

| | | |
|---|---|---|
| `A ((...))((...))` | `A ((...))((..---.))--` | `(([[[))((]]]))` |
| `B ..(((....))).. ` | `B ..-----(((....)))..` | `aaggguuaacccuu` |
| `A aaggguuaacccuu` | `A aaggguuaacc---cuu--` | |
| `B aaggguuaacccuu` | `B aa-----ggguuaacccuu` | |

*Table 18: a toy example showing a limitation of forest alignment. Left: two hypothetical RNA secondary structures, A and B. Middle: RNAForester alignment. Although the two structures have identical sequences, the alignment creates artifactual indels in order to force the two structures into a single ordered forest, no matter the scoring function. Right: in theory, A and B could be represented as a tertiary structure.*

In summary, current RNA structure alignments are not natural representations for the similarities and variations of arbitrary RNAs, and have strong structural constraints that give structural similarity precedence over sequence similarity regardless of the scoring function. From the last example, we might expect that our limitation is that our alignments are themselves secondary structures. Perhaps a secondary structure alignment should be represented by some kind of tertiary structure profile. I will make a stronger claim: the problem is that RNA structure alignments are inherently partially ordered, and they cannot be represented faithfully under a total ordering. There are three main arguments in support of this claim.

First, (Lee *et al.*, 2002) make a compelling argument that sequence alignments are better represented with a partial ordering. Since a RNA secondary structure alignment essentially contains an alignment of sequences, we should expect a partial-order representation to be just as beneficial for structures. Therefore we expect it to provide a non-degenerate representation for insertions and deletions, multiple alignments that consider the optimal alignment of each structure against each other structure, and a compressed representation that results in faster alignments.

In addition, an RNA structure alignment should ideally tell us something like "the sequences are similar here, the structures are similar there, and this is how variations in sequence produce variations in structure". This is a second sense in which we expect a partial ordering to improve RNA alignment. The structural constraints inherent in existing alignment methods make them give structural features first priority; later, they make up for any discrepancies by creating sequence indels. This makes it hard to analyze the alignments in an evolutionary context, since there is no clean separation between sequence and structure variation. Partial order alignments can represent 'one path *or* the other', so perhaps they can represent 'sequence *or* structure' similarity.

There is a third, stronger sense in which a partial-order representation is natural for RNA structure alignments. Let us consider base pair shifts, a relatively common structural variation. Base pair shifts happen when a base switches to a different pairing partner, often because its original pairing partner was substituted or deleted.

The table below shows a segment of an arc-annotated string alignment of RNAse P RNAs, as seen in (Wang and Zhang, 2001). RNAse P, in concert with a protein cofactor, has a vital role in the processing of transfer RNAs in bacteria, archaea, and eukaryotes. To carry out its function, RNAse P relies on several conserved structural features. The center region of this alignment, for example, has a long stem common to both RNAs despite sequence variations. For example, two C-G base pairs in *A. eutrophus* become G-U and G-C base pairs in *S. bikiniensis*. These compensatory base mutations are an indication that the structure has been conserved by evolution; if the structure was not conserved, we would see random mutations that could change its structure.

| arc-annotated alignment (interpretation A) | manual alignment (interpretation B) |
|---|---|
| `..-((((.((((((((....)))))))).))--)).` | `..((((.((((((((....)))))))).)))).` |
| `...(((--((((((((....))))))))--)..)).` | `...((-(((((((((....)))))))))..)).` |
| `AG-UCUUGCCGCCGGGUUCGCCCGGCGGGAA--GGG` | `AGUCUUGCCGCCGGGUUCGCCCGGCGGGAAGGG` |
| `AGACCG--CCGGGGACCUCGGUCCUCGG--UAAGGG` | `AGACC-GCCGGGGACCUCGGUCCUCGGUAAGGG` |

*Table 19: segment of pairwise alignments of A. eutrophus (top rows) and S. bikinensis (bottom rows) RNAsePs. Left: arc-annotated alignment from (Wang and Zhang, 2001). Right: an alternate interpretation of the evolutionary relationship between the two RNAs.*

The sides of the alignment are more difficult to interpret. The arc-annotated alignment inserts/deletes seven bases in order to map both structures onto the same consensus set of arcs. In this view, despite the many indels and base substitutions, only one base pair has changed; this part of the structure must have several conserved base pairs that are very resistant to mutations. I will call this *interpretation A*.

Note that interpretation A leaves two adjacent 'AA' regions mismatched or gapped. If we give enough weight to sequence information, we might allow such similar regions to influence our structural alignment. In the lower table, I have constructed an alignment of this type which I will call *interpretation B*. In this view there has only been a single base indel, but it has combined with a few substitution mutations to remove two base pairs and to shift several others. The 'U' of an 'A'-'U' pair has mutated to a 'C'; as a consequence, the new 'C' pairs with a 'G' instead, and the 'G's old partner has shifted its bonds to a different 'G'.

Interpretations A and B are both informative; they represent two plausible functional/evolutionary relationships between the RNAs. Ideally, our scoring function will determine which interpretation we choose. However, arc-annotated alignments impose a single consensus structure on their RNAs, so they cannot generate interpretation B. Forest alignments allow a wider variety of interpretations. For example, they can partially align the two 'AA' regions; a forest allows an 'A' to be unpaired in one structure but paired in another. `RNAForester` can actually generate interpretation B during its alignment phase, but fails to reconstruct it during its trace back. Each base node in an RNA forest must have only a single partner, so base pair shifts are verboten.

In fact, base pair shifts can only be represented under a partial ordering. During alignment, we need to say “match with this pairing partner *or* that pairing partner”, whereas a total order alignment only has a notion of the 'next' base or pair. As a preview of things to come, the table below shows how RNA partial-order alignments (RNAPOAs) represent interpretations A and B. The sequence data in a RNAPOA is just like a normal PO-MSA. Structural data is held in 'pair' nodes that connect pairs of nucleotides. One pair node can connect to more than one pair of bases; for example, it might represent a “G-U” pair *or* a “C-G” pair. Base pair shifts result in some bases that are connected to more than one pair node; in future alignments, those bases can choose either pairing partner.

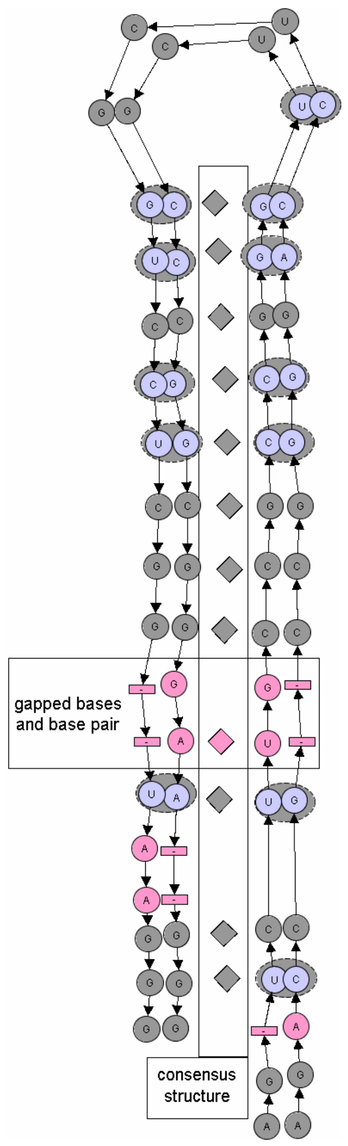
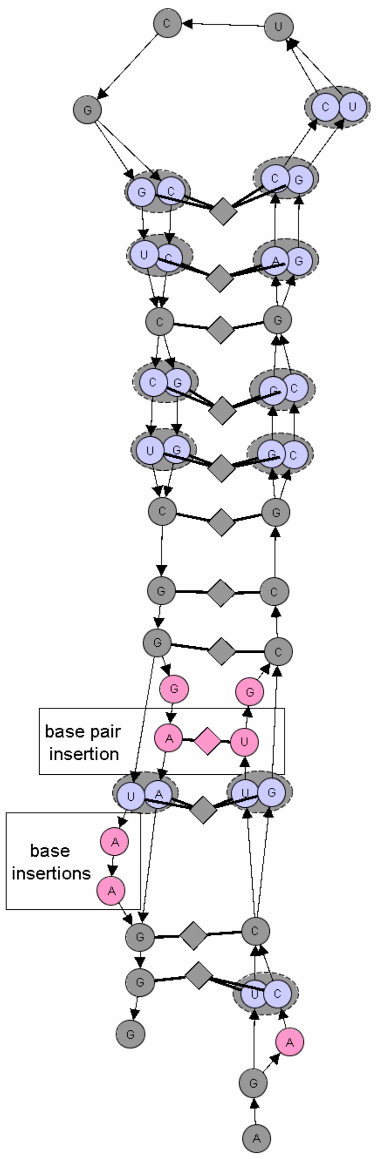
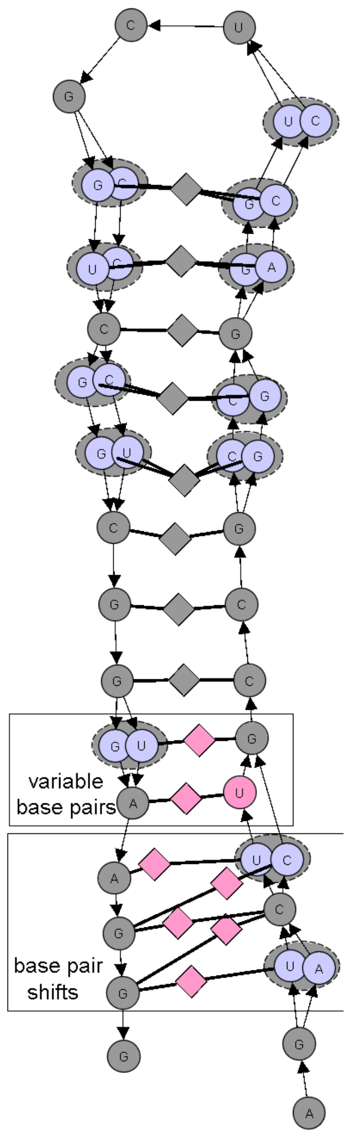
| arc-annotated alignment (interpretation A) | RNAPOA (interpretation A) | RNAPOA (interpretation B) |
|---|---|---|
|  |  |  |

*Table 20: comparison of partial-order and total-order alignments. Indels are drawn in pink. Matched positions are drawn in gray. Mismatched bases are drawn in light blue and circled in gray. Interpretation A adds indels to maximize the number of base pairs in common; in this view, a spate of insertion and deletion mutations left the secondary structure largely intact. Interpretation B breaks base pairs to match similar nucleotides; in this view, a small number of mutations created new base pairs and shifted existing ones. Both interpretations are informative, but total-order alignments cannot represent base pair shifts and hence can only generate interpretation A. Partial order alignments can produce either interpretation given the appropriate scoring function.*

Before I go any further, I should mention two important caveats. First, in this thesis I consider RNAs with known structure. One might argue that RNA structure alignment *per se* is not the right problem to consider. A major use for structure alignments is to predict the structures of phylogenetically and/or functionally related RNAs. However, it is difficult to determine RNA structures experimentally, so most large datasets of RNA structures are generated by structure prediction programs. In that case, it would be best to explicitly combine structure prediction and alignment, (e.g. Sankoff, 1985; Hofacker *et al.*, 2004), rather than treating them as separate steps. In the Discussion section, I will give some thoughts on integrating partial order RNA alignments with structure prediction methods. In particular, RNAPOAs turn out to be a generalization of the method of (Hofacker *et al.*, 2004) but have better efficiency in some cases.

Second, partial ordering is best used in situations in which we need a detailed representation of the variation within a set of RNAs. The main applications that I envision are: 1) evolutionary analysis of structural RNAs, and 2) exploring the variation within alternate folds generated by structure prediction algorithms. There are other contexts in which a partial-order formulation isn't as useful. For example, if we are searching a large database with a structured RNA as a query, we are not searching for sequence *or* structure similarity, but rather sequence *and* structure similarity. Partial-order queries would be a bit more flexible for pattern-matching than total-order queries. However, if we use alignment scores to testing the statistical significance of our search results, the permissive nature of partial-order alignment might actually be an impediment.

# 3. RNA partial-order alignments (RNAPOAs)

### 3.1 RNAPOAs

An RNAPOA is a directed acyclic graph. For RNAs with no base pairs, an RNAPOA is the same as a partial-order sequence alignment. Each node $s_i$ represents a nucleotide. $s_i$ has has a list of predecessors *prev*($s_i$) and a list of successors *next*($s_i$).

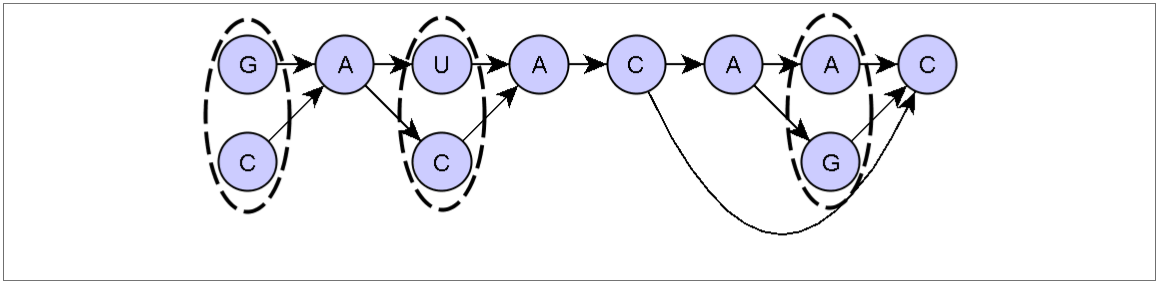


*Table 1: an RNAPOA without any base pairs is the same as a partial order sequence alignment.*

Previously, we were able to discuss PO-MSAs in terms of suffixes only, but now we'll have to talk about 'subsequences' as well. Subsequences are defined in terms of a total ordering, so we'll have to use a new definition. For an RNAPOA *S*, I will assign each pair of nodes ($s_i$, $s_j$) one of four possible partial orderings:

| | |
|---|---|
| 1. $s_i \leq s_j$, | if we can reach $s_j$ from $s_i$ using *next*, |
| 2. $s_j \leq s_i$, | if we can reach $s_i$ from $s_j$ using *next*, |
| *3*. $s_i \leq s_j$ and $s_j \leq s_i$, | if $s_i$ and $s_j$ are the same. |
| *4*. $s_i$ and $s_j$ are incomparable, | if neither can be reached from the other via *next*. |

Given those definitions, let ($s_i$, $s_j$) define a *span* - a set of all $s_k$ such that $s_i \leq s_k \leq s_j$. A span of a partial-order alignment is analogous to a subsequence in a sequence alignment. For example, in the above sequence-only RNAPOA, a span from the first “C” to the last “C” includes every node except for the first “G”. A span from the last “A” to the last “C” only includes those two bases.

For RNAs with base pairs, I will define two separate types of RNAPOAs: string-RNAPOAs, based on arc-annotated strings, and forest-RNAPOAs, based on ordered forests. The two types are quite similar, but there are subtle differences in their editing operations and alignment complexity which make one or the other preferable for specific purposes.

A string-RNAPOA is analogous to an arc-annotated string, only it allows each base to be involved in more than one base pair. Each base node $s_i$ is has a set of pairs *pair*($s_i$). We can draw the RNAPOA with 'pair' nodes that have bidirectional edges to their 5'-most and 3'-most bases.

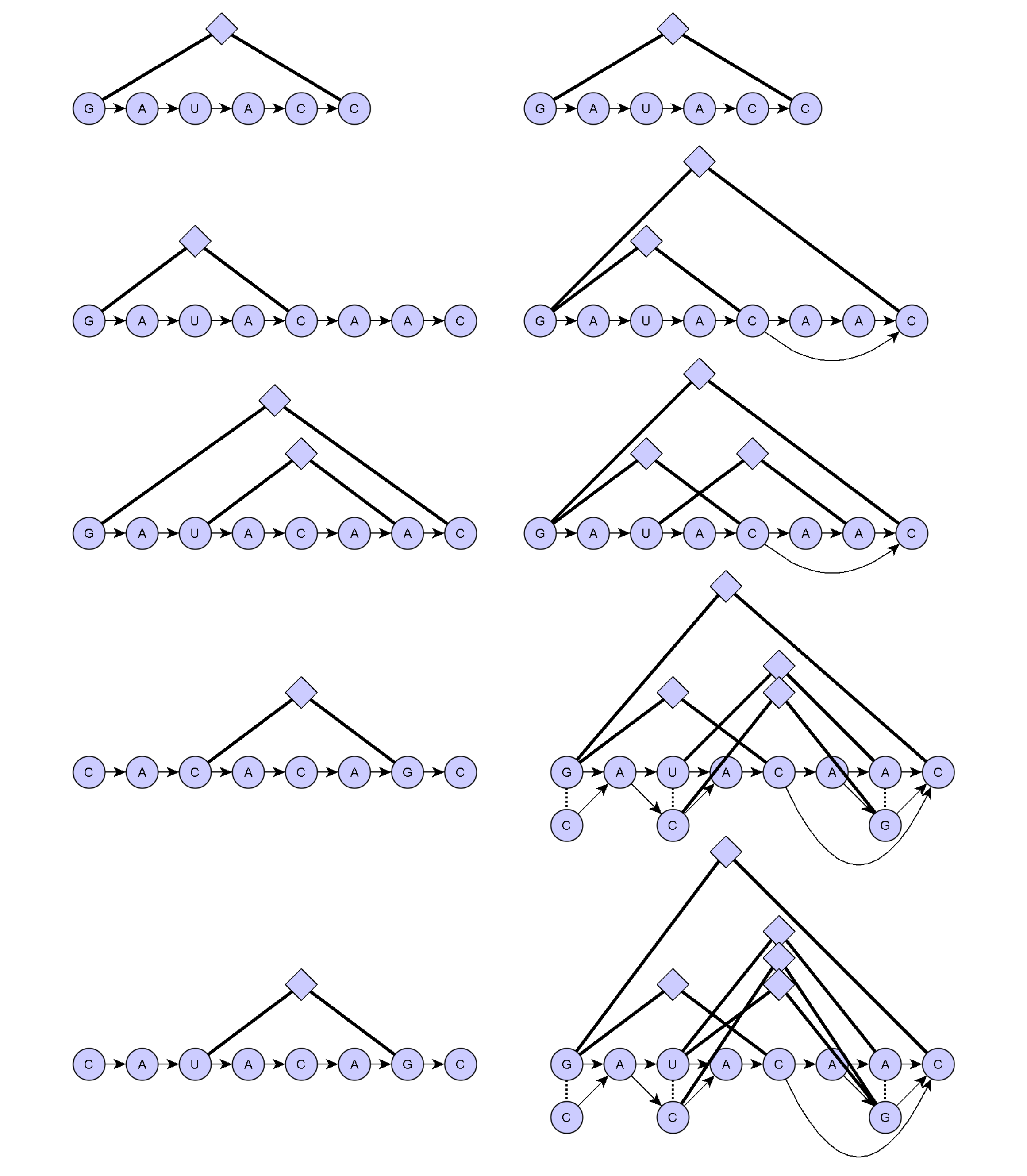


*Table 2: merging individual RNA structures (left) one-by-one into a string-RNAPOA (right). Note: in this image, aligned bases are represented by dashed lines instead of dashed circles. After the first merging, we see that the RNAPOA can represent base pair shifts; "G" can be bound to either "C". The second merging shows how string-RNAPOAs represent crossing bases. The third merging shows the representation for alternate base pairs - the structures can have either a "U-A" bond or a "C-G" bond. After the final merging, the RNAPOA also allows a "U-G" bond.*

What would happen if we placed a pair node between every pair of positions in the RNAPOA? Then, for a suitably defined scoring function, it would be analogous to the pairing probability matrices of (Hofacker *et al.*, 2004). However, RNAPOAs can represent individual structural variation, not just a consensus. For example, we could say “the probability is 1.0 if ($s_i$, $s_j$) = (C, G), but but the probability is 0.6 if ($s_i$, $s_j$) = (U, G)”.

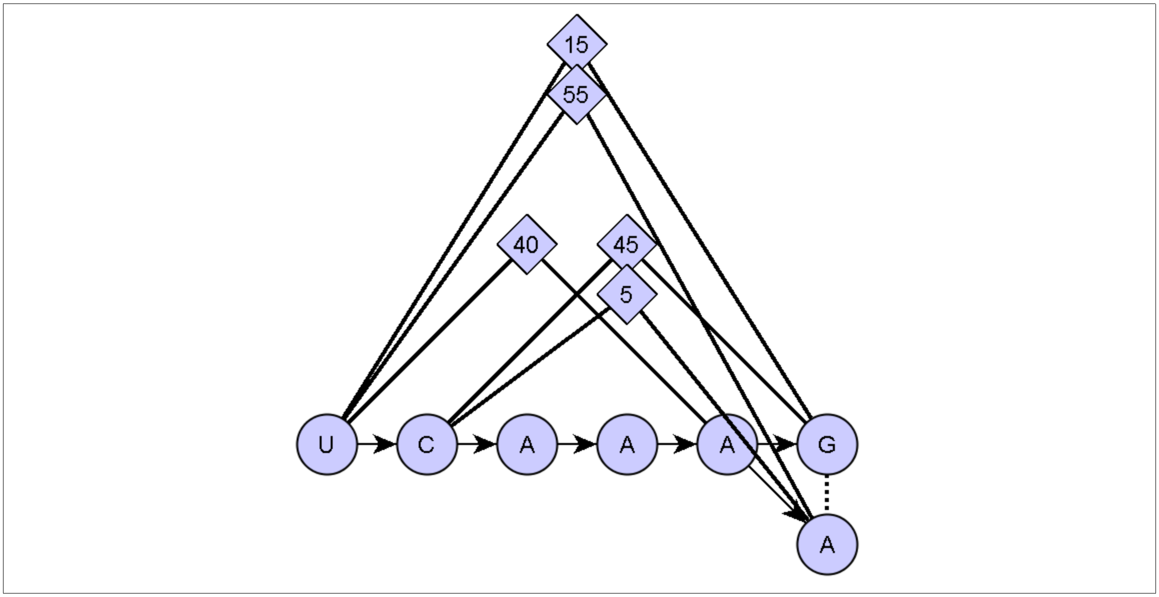


*Table 3: string-RNAPOA representing base pairing probabilities. RNAPOAs can represent the changes in base pairing probability due to point mutations; if there is a G-to-A mutation, it changes the probability of pairing with U.*

A forest-RNAPOA is analogous to an ordered forest, only it allows each pair node to have multiple pair 'subforests'. Because of the partial ordering of next and pair edges, a forest-RNAPOA need not look like a forest per se. For example, an alignment of ordered forests can't generate tertiary structure, but a forest-RNAPOA can represent crossing base pairs:

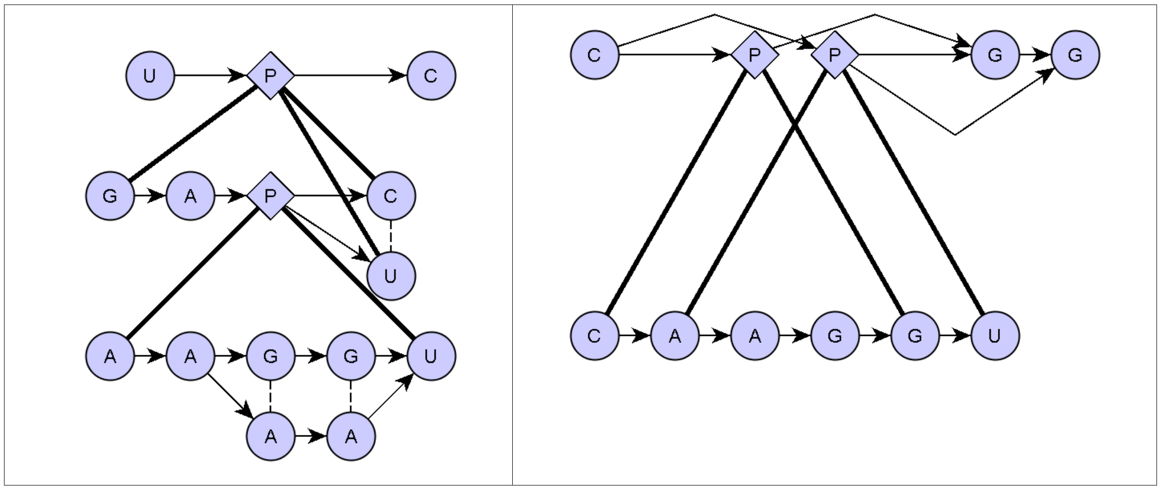


*Table 4: forest-RNAPOAs. Left: just like a forest, but with variable structure. Note that each base pair can represent more than one base pair; here, the top pair represents either a "G-C" or "G-U" pair. Right: forest-RNAPOAs can represent sequence indels, pair indels, and crossing base pairs.*

If we aligned a plain sequence to a forest-RNAPOA, we get something that looks very much like a string-RNAPOA. In fact, I will call this sort representation 'stringlike'. An ordinary forest forces us to visit each pair node during a traversal of the graph. In contrast, all base pairs are optional in a 'stringlike' RNAPOA. This makes *MaxLoop*(*F*) = |*F*|, removing the main efficiency advantage of forest alignment. However, stringlike RNAPOAs are sometimes useful. As we'll see later, they will allow us to compute alignments that include crossing base pairs with reasonable efficiency.

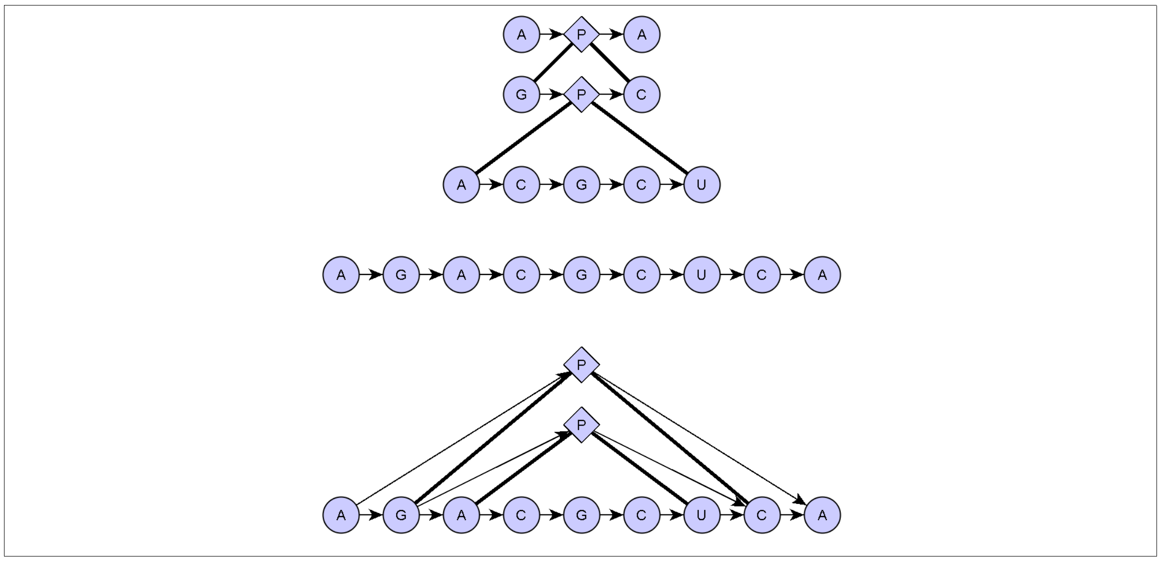


*Table 5: 'stringlike' forest-RNAPOAs. We can create them by aligning a forest-RNAPOA (top) with each sequence that it contains (middle), producing an RNAPOA in which all base pairs are optional paths.*

If we go further and add more pair nodes, and we'll again get something like an RNA base pairing probability matrix. However, this greatly increases the number of nodes in *F*. Since forest alignments have to consider suffixes and non-suffixes starting at each node in *F*, they have very high time and space complexity for this case. As we'll see, forest-RNAPOAs are not suitable for probabilistic alignments.

## 3.2 Optimal alignment of two RNAPOAs

Here I will develop algorithms for optimal alignment of string-RNAPOAs and forest-RNAPOAs. I base the string-RNAPOA alignment algorithm on the arc-annotated alignments of (Wang and Zhang, 2001). The algorithm takes essentially $O(Bases(S)Pairs(S)Bases(T)Pairs(T) + Bases(S)^2 + Bases(T)^2)$ time and space, plus a few extra factors that grow slowly when the structures contained in the RNAPOAs are dissimilar. I will show that when the number of pairs is large, string-RNAPOA alignment becomes analogous to the base-pairing probability alignment algorithm of (Hofacker *et al.*, 2004).

In the second subsection, I will discuss forest-RNAPOA alignment. forest-RNAPOAs can be aligned more efficiently than string-RNAPOAs in certain cases, but have some of the same limitations as ordered forests. That is, they do not generate alignments that have crossing base pairs. However, we can work around this problem. I show that if one of the forest-RNAPOAs being aligned has a certain 'string-like' form, we can still account for crossing base pairs.

### 3.2.1 string-RNAPOA alignment

Recall the approach of (Wang and Zhang, 2001) for arc-annotated alignment. In that scheme, we used four separate recurrences - *ALIGN*(base,base), *ALIGN*(pair,base), *ALIGN*(incompatible pair, incompatible pair), and *ALIGN*(compatible pair, compatible pair). If we want to handle RNAPOAs, we can't special-case these situations; each node can have several different pairs, or may be paired in some aligned structures and unpaired in others. We will have to consider all of these cases in the same recurrence relation.

Let us consider an alignment of two spans, $(s_i,s_j)$ and $(t_k,t_l)$. The base replacement, base insertion, and base deletion operations are the same as in partial-order alignment of sequences - we maximize over all *next* spans. The pair replacement operation is more complicated; I explain it in detail below.

```
$ALIGN((s_i,s_j),(t_k,t_l))$ = max of
base replace    $base\_score(s_i,t_k)$ + max over $s_{next}$ in $next(s_i)$, $t_{next}$ in $next(t_k)$ {
                                    $ALIGN((next(s_i),s_j), (next(t_k),t_l))$
                                    }
base insert     $base\_score(-,t_k)$ + max over $t_{next}$ in $next(t_k)$ {
```

```
                                   $ALIGN((s_i,s_j), (t_{next},t_l))$
                                   }
base delete     $base\_score(s_i,-)$ + max over $s_{next}$ in $next(s_i)$ {
                                   $ALIGN((s_{next},s_j), (t_k,t_l))$
                                   }
pair replace    max over $(s_i,s_m)$ in $pair(s_i)$, $(t_k,t_n)$ in $pair(t_k)$ {
                    $pair\_score((s_i,s_m),(t_k,t_n))$
                    + max over $s_{next}$ in $next(s_i)$, $t_{next}$ in $next(t_k)$ {
                        $ALIGN_{ignore\_end}((s_{next},s_m), (t_{next},t_n))$
                    }
                    + max over $s_{next}$ in $next(s_m)$, $t_{next}$ in $next(t_n)$ {
                        $ALIGN((s_{next},s_j),(t_{next},t_l))$
                    }
                }
```

where $ALIGN_{ignore\_end}((s_i, s_j), (t_k, t_l))$ is just like *ALIGN*, except it ignores the last base in each span. In other words, if $s_i = s_j$, then it sets $(s_i,s_j) = \emptyset$, and if $t_k = t_l$, then it sets $(t_k,t_l) = \emptyset$.

What happens during the pair replace edit operation? Unlike arc-annotated alignments, a base may have many alternate pairs. So $pair(s_i)$ isn't just a single base pair $(s_i,s_m)$ or $(s_m,s_i)$, but rather a set of such base pairs. We have to maximize over many pair replacements - all replacements of base pairs that start with $s_i$ and $t_k$. Furthermore, for each set of pairs $(s_i,s_m)$ and $(t_k,t_n)$, we have to consider many different subalignments. In arc-annotated alignment, a pair replacement divides the spans into exactly two subalignments: an alignment of the bases 'inside' the pair - $(next(s_i), prev(s_m))$ with $(next(t_k), prev(t_n))$ - and an alignment of the bases 'outside' the pair - $(next(s_m), s_j)$ with $(next(t_n), t_l)$. In string-RNAPOAs, pair replacement is not so simple. Each node can have many *next*s and *prev*s, so we have to maximize over many possible 'inside' and 'outside' spans.

This introduces two dangers. First, not all of those potential inside and outside spans are valid subalignments of $(s_i,s_j)$ and $(t_k,t_l)$. For example, we shouldn't consider $(s_{next}, s_j)$ if there isn't any path from $s_{next}$ to $s_j$. We also have to make the same validity checks on base pairs that we did for arc-annotated alignment; we shouldn't consider a base pair $(s_i,s_m)$ if $s_m$ isn't between *si* and $s_j$. Second, for the 'inside' subalignment, we have to optimize over many different $prev(s_m)$ and $prev(t_n)$ nodes. So for each base pair $(s_i,s_m)$ such that $prev(s_m) = \{s_1, s_2, ...\}$, a naive approach would compute and store many separate sets of subalignments - subalignments ending at $s_1$, subalignments ending at $s_2$, etc.

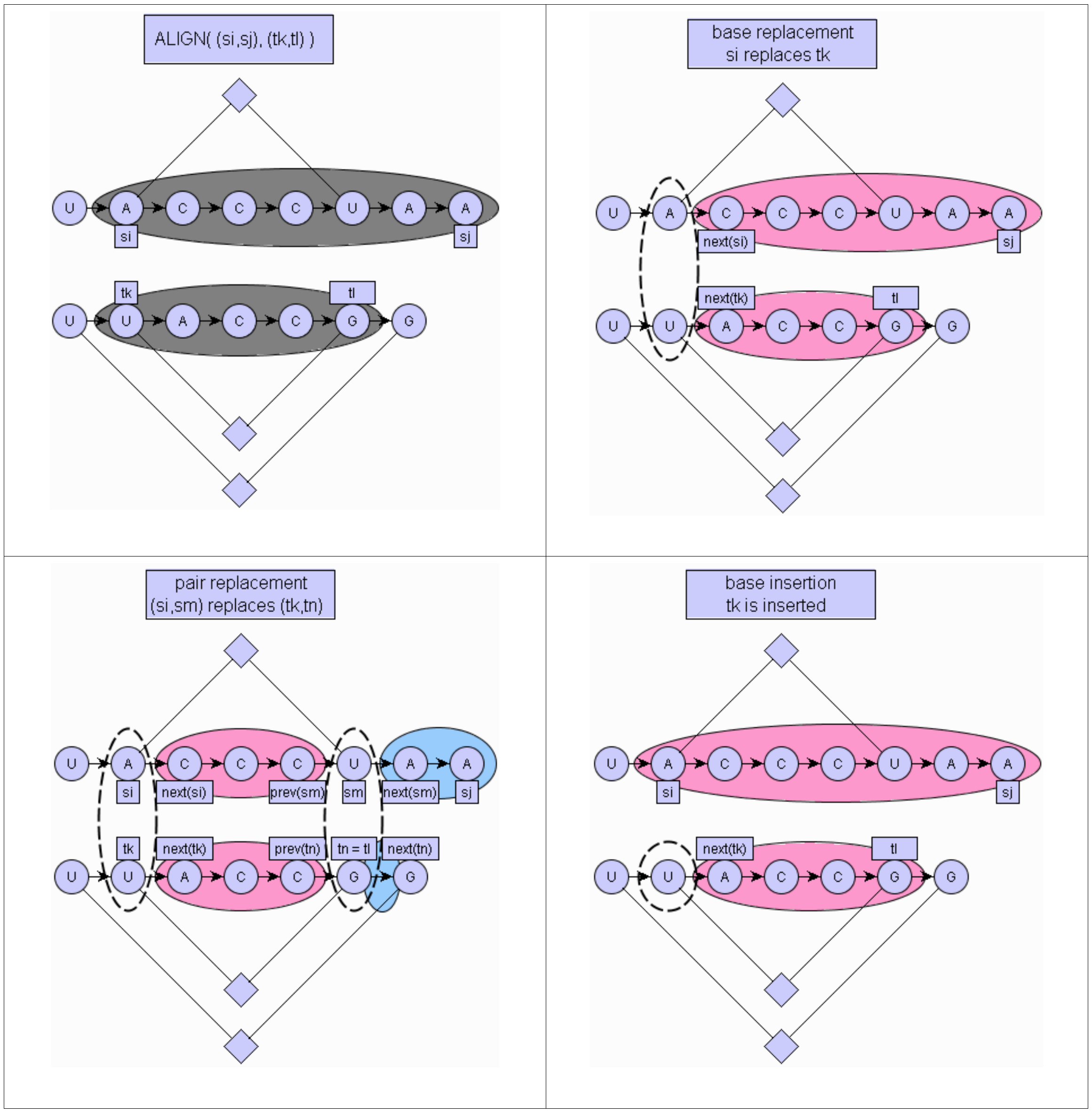


*Table 6: string-RNAPOA alignment edit operations. For simplicity, each RNAPOA represents only a single structure; in general, there can be more than one next, prev, and pair node for each base. The spans to be aligned are gray. Aligned nodes are enclosed by dashed circles. Subalignments are pink or blue. Top left: alignment of the spans ($s_i$,$s_j$) and ($t_k$,$t_l$). Top right: base replacement. Bottom right: base insertion. Bottom left: pair replacement, which depends on two subalignments: one with spans 'between' the paired bases, and another with spans 'outside' the paired bases. Note that since tn = tl, the blue span (next($t_n$),$t_l$) is empty, so the 'outside' subalignment will have to be a deletion of the other blue span (next($s_m$),$s_j$).*

To solve the first problem, we want to be able to quickly test a span or pair's validity.One way to do this is to simply precompute the partial-ordering '≤' between each pair of nodes in *S* and store the results in a matrix $S_{\leq}$. For example, in my implementation,

$$S_{\leq}[i][j] = \begin{cases} 1, & \text{if there is a path from } s_i \text{ to } s_j \text{ using only } \textit{next} \text{ edges} \\ 0, & \text{otherwise.} \end{cases}$$

We can then compute the validity of a span or base pair with a few constant-time lookups. For example, $(s_{next}, s_j)$ is invalid if $S_{\leq}[next][j] = 0$. We can ignore any base pair $(s_i,s_m)$ if $S_{\leq}[i][m] = 0$ (i.e. when $s_m$ comes before $s_i$) or if $S_{\leq}[m][j] = 0$ (when $s_m$ comes after $s_j$, or when $s_m$ and $s_j$ are on alternate paths).

To solve the second problem, we want to avoid computing separate subalignments for each predecessor node in $prev(s_m)$. That's why we use $ALIGN_{ignore\_end}$. It pretends that $(s_j,s_j) = (t_l,t_l) = \varnothing$, so it implicitly maximizes over all *prev* nodes of $s_j$ and $t_l$:

```
$ALIGN_{ignore\_end}((s_{next}), s_m), (t_{next}, t_n)) =$
    max over $s_{prev}$ in $prev(s_m)$, $t_{prev}$ in $prev(t_n)$ {
        $ALIGN((s_{next}, s_{prev}), (t_{next}, t_{prev}))$
    }
```

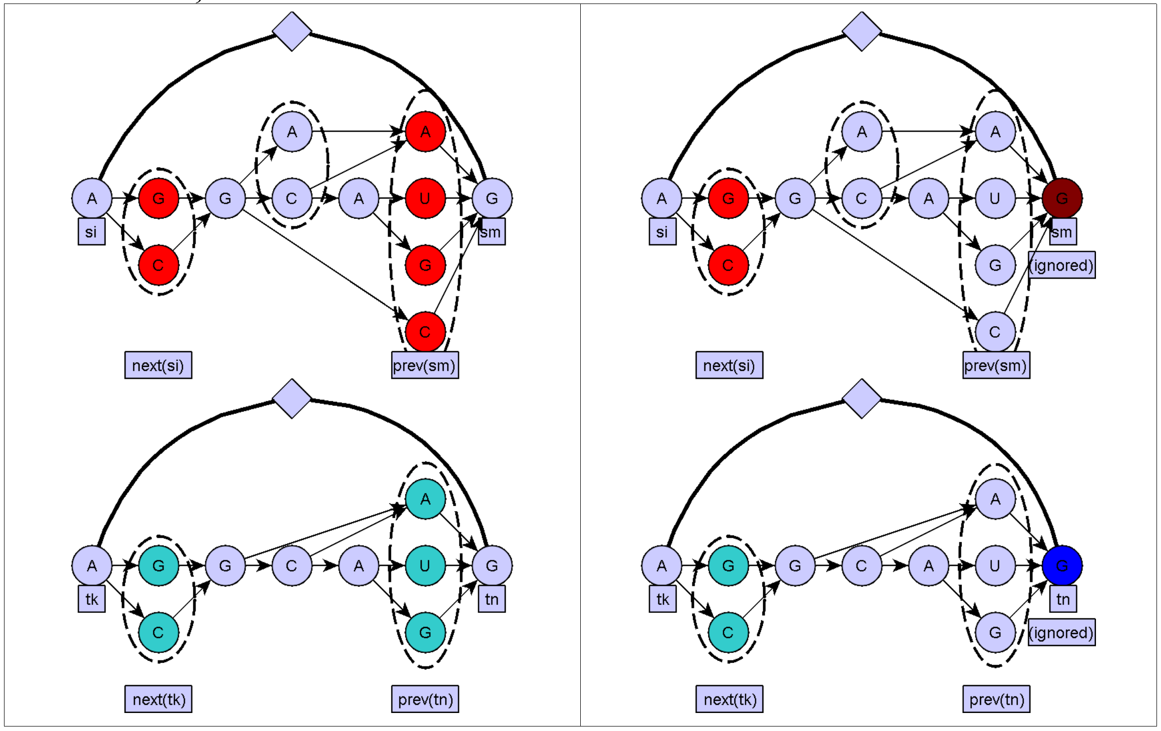


*Table 7: Left: base replacement of (si,sm) with (tk,tn). We have to optimize over all (2x4)x(2x3) combinations of next and prev nodes (bright colored) in S and T. Hence we compute and store many different subalignments. Right: $ALIGN_{ignore\_end}$ does this implicitly, but optimizes over fewer combinations and only computes subalignments of spans ending in an ignored $s_m$ and an ignored $t_n$ (dark colored).*

We compute local/global alignments and merge *S* and *T* together in the same way that we did for PO-MSAs. The only difference is that we also have to decide how to merge pair nodes: we merge all pair nodes which were aligned by pair replacement operations. The resulting RNAPOA combines the properties of PO-MSA and arc-annotated alignments: it is the optimal 'threading' of any one secondary structure and sequence of *S* with any one secondary structure and sequence of *T*.

What is the algorithm's complexity? We'll need some definitions.

*Bases(S),* the number of base nodes in *S*.
*Pairs(S),* the number of pair nodes in *S*.
*BaseDeg(S),* the max number of *next* nodes from any base in *S*.
*PairDeg(S),* the max number of pairs starting at any base in *S*.

Trivially, $Pairs(S) < Bases(S)^2$ and $PairDeg(S) < Bases(S)$. Also, we can generally consider *BaseDeg(S)* to be constant. Intuitively, $s_i$ shouldn't have more than one successor with the same nucleotide. If it did, then an optimal alignment would have matched those identical successors and merged them together into a single node. So we expect *BaseDeg(S)* to be at most $|\{a,u,g,c\}| = 4$. This should be true for most real-world alignments of homologous RNAs. It is analogous to the common assumption that the number of gaps in a total-order alignment won't grow very quickly. Unfortunately, as I will discuss in section 3.3, the partial order formulation of (Grasso and Lee, 2004) that I build on in this thesis doesn't actually guarantee this property. I will have to keep track of *BaseDeg* during these derivations, which will produce slightly ugly expressions; I will also give simplified expressions to show the algorithm's likely real-world behavior.

As in arc-annotated alignments, we start by considering all $O(Bases(S)Bases(T))$ alignments of suffixes. Also, pair replacement operations create 'pair suffixes' - spans $(s_i, s_m)$ that we align using the $ALIGN_{ignore\_end}$ function. For each pair $(s_k,s_l)$ in *S*, there are at most $O(Bases(S))$ pair suffixes. Therefore we need to store $O(Bases(S)Pairs(S)Bases(T)Pairs(T))$ subalignments. In addition, we spend $O(Bases(S)^2 + Bases(T)^2)$ time and space to compute and store the partial-ordering matrices $S_{\leq}$ and $T_{\leq}$. Hence the space complexity is $O(Bases(S)Pairs(S)Bases(T)Pairs(T) + Bases(S)^2 + Bases(T)^2)$.

Actually, we can give a better bound. The worst case is when the RNAPOA contains all possible base pairs. In that case $Pairs(S) = O(Bases(S)^2)$, but there are still only $O(Bases(S)^2)$ spans in *S*, so there are only $O(Bases(S)^2Bases(T)^2)$ subalignments. Thus the space complexity is really:

$$O(Bases(S)^2 + Bases(T)^2 + Bases(S)Bases(T) * \min(Pairs(S), Bases(S))\min(Pairs(T), Bases(T))).$$

For each subalignment we have to consider four edit operations. It is easy to see their time complexity:

| ***Edit Op.*** | ***Edit Operation Time Complexity*** |
|---|---|
| base replace | $O(\mathit{BaseDeg}(S)\mathit{BaseDeg}(T))$ |
| base insert | $O(\mathit{BaseDeg}(T))$ |
| base delete | $O(\mathit{BaseDeg}(S))$ |
| pair replace | $O(\mathit{PairDeg}(S)\mathit{PairDeg}(T)\mathit{BaseDeg}(S)\mathit{BaseDeg}(T))$ |

Which gives the algorithm the following time complexity:

$$\begin{aligned}O(&\mathit{Bases}(S)^2 + \mathit{Bases}(T)^2 + \\ &\mathit{Bases}(S)\min(\mathit{Pairs}(S), \mathit{Bases}(S)) \\ &* \mathit{Bases}(T)\min(\mathit{Pairs}(T), \mathit{Bases}(T)) \\ &* \mathit{PairDeg}(S)\mathit{PairDeg}(T)\mathit{BaseDeg}(S)\mathit{BaseDeg}(T))\end{aligned}$$

For individual RNAs, $O(\mathit{BaseDeg}(S)) = O(\mathit{PairDeg}(S)) = O(1)$, so the algorithm requires $O(\mathit{Bases}(S)\mathit{Bases}(T)\mathit{Pairs}(S)\mathit{Pairs}(T) + \mathit{Bases}(S)^2 + \mathit{Bases}(T)^2)$ time and space. This is essentially the same as an arc-annotated string alignment. When we add many structures to a RNAPOA, *BaseDeg*, and *PairDeg* will increase. As I argued earlier, for most practical purposes we are interested in RNAs that are at least somewhat similar to each other, so these factors will not grow too quickly.

What if we want to consider all possible base pairs? Then this algorithm becomes a partial-order version of the method of (Hofacker *et al.*, 2004). The complexity is equivalent: $O(\mathit{Bases}(S)^2\mathit{Bases}(T)^2)$ space and $O(\mathit{Bases}(S)^3\mathit{Bases}(T)^3)$ time. In practice, most base pairing probabilities are very low, so we can do better. First of all, there are sixteen possible base pairs in {a,u,g,c} x {a,u,g,c}, but only six of them are common.[19] Most RNAs have relatively equal proportions of each of nucleotide, so this greatly reduces the number of probable pairs. Second, bases that are too near to each other cannot be paired due to bond angle constraints. Third, each base's total pairing probability cannot be above 1.0, so bases with a few highly favored folds will be unlikely to pair elsewhere. For example, bases in the middle of a conserved hairpin stem-loop are very unlikely to pair anywhere else. If we only add pair nodes for probable base-pairs (defining 'probable' with some reasonable threshold), we should reduce the algorithm's time and space requirements by a large factor.

It is important to note that string-RNAPOAs only allow three types of editing operations: base replacement, base indels, and pair replacement. If we want to delete two paired bases, we have to delete them individually; this is less flexible than arc-annotated

19 (a,u), (u,a), (g,c), (c,g), (g,u), and (u,g)

alignment, in which we can define a separate score for simultaneous insertion/deletion of two paired bases. Unfortunately, we can only do that efficiently if each base has only a single associated structure. In that case, if we insert/delete a base pair, we only need to gap its first base; any valid subalignment is forced to gap the second base as well. This is why (Wang and Zhang, 2001) get away with the trick of allocating (1/2)**pair_score*($(s_i,s_j),(-,-)$) to each base. In our case, each base has several alternate base pairs. If we choose a subalignment that gaps $s_i$, we cannot assure that it also gaps $s_j$.

Typical scoring models for sequence alignment assume that indels occur when several consecutive bases are inserted or deleted. In this view, there is no reason why two separate indels should co-occur. However, the prevalence of compensatory base substitutions tells a different story: evolution culls RNA mutations that disrupt important structural features. Indeed, structural RNAs often have simultaneous pair indels inside conserved stems (e.g. see (Holmes, 2004)), although linear indels are also possible, as in 'interpretation B' of the alignment I discussed in section 2.6.3.

If we want to allow both base indels and base pair indels, it comes at a cost.[20] Recall that pair indels in forest alignment divide a structure into two parts. We then have to optimize over all ways to split the other structure into two parts. The same is true here. Say we are aligning the spans $(s_i,s_j)$ and $(t_k,t_l)$, and we want to delete the base pair $(s_i,s_m)$. To guarantee that both $s_i$ and $s_m$ are deleted simultaneously, we have to divide $(s_i,s_j)$ into sub-spans that don't contain either $s_i$ or $s_m$. - the spans 'inside' and 'outside' the base pair. Which parts of $(t_k,t_l)$ should we align with the 'inside' and 'outside'? We don't know! As in forest alignment, we have to optimize over all possible 'split nodes' to divide $(t_k,t_l)$ into two parts.[21]

```
pair delete     max over $(s_i,s_m)$ in $pair(s_i)$ {
                        $pair\_score((s_i,s_m),(-,-))$
                        + max over $t_{split}$ in $(t_k,t_l)$ {
                                max over $s_{next}$ in $next(s_i)$ {
                                        $ALIGN_{ignore\_end}((s_{next},s_m), (t_k,t_{split}))$
                                }
                                + max over $s_{next}$ in $next(s_m)$ {
                                        $ALIGN((s_{next},s_j),(t_{split},t_l))$
                                }
                        }
                }
```

Combining all of these maximization steps, finding the pair deletion with the

20 By the way, we could modify arc-annotated alignment to use single base insertions/deletions against paired bases. Unfortunately, then it couldn't efficiently handle pair indels; if it allows single base indels, it can't guarantee that simultaneous indels of paired bases will gap both bases of the pair.

21 Again, the notation here doesn't show it, but we also have to try splits where one side of the split is an empty span and the other side is just the full $(t_k,t_l)$.

highest score takes $O(PairDeg(S)Bases(T)(BaseDeg(S) + BaseDeg(T)))$ time. Pair insertion is just a mirror of pair deletion, with all the *S* and *T* terms switched. Note that since we have to try all arbitrary splits of *S* and *T*, we will have to align all spans of *S* and *T*. Hence our space complexity will increase to $O(Bases(S)^2Bases(T)^2)$, and the total alignment time will increase to:

$$\begin{aligned} O(&Bases(S)^2Bases(T)^2 \\ &(PairDeg(S)PairDeg(T)BaseDeg(S)BaseDeg(T) \\ &+ PairDeg(T)Bases(S)(BaseDeg(T) + BaseDeg(S)) \\ &+ PairDeg(S)Bases(T)(BaseDeg(S) + BaseDeg(T))) \end{aligned}$$

The last three terms in this expression are for pair replacement, pair insertion, and pair deletion operations, respectively. For most real-world cases, the complexity should be:

space: $O(Bases(S)^2Bases(T)^2)$

time: $O(Bases(S)^2Bases(T)^2$
$* (PairDeg(S)PairDeg(T) + Bases(S)PairDeg(T) + PairDeg(S)Bases(T))$

If *S* and *T* are individual RNA structures or RNAPOAs with few alternate paths, the running time is quintic: $O(Bases(S)^2Bases(T)^2(Bases(S)+Bases(T)))$. For RNAPOAs with all possible base pairs, the running time is again $O(Bases(S)^3Bases(T)^3)$.

### 3.2.2 forest-RNAPOA alignment

Next I will consider forest-RNAPOAs. I will base my alignment algorithms on those of (Jiang *et al.*, 1995; Jansson *et al.*, 2004). First, however, I will soup them up to handle the edit operations *base pair replacement* and *base pair indels* for the forest representations of (Hochsmann *et al.*, 2003). Hochsmann *et al.* define operations on pair bonds, not on base pairs. To straighten out my terminology: editing operations on *pair bonds* only replace/indel an individual pair node. Operations on *base pairs* replace/indel the pair node and its paired bases at the same time. See the table below for a comparison.

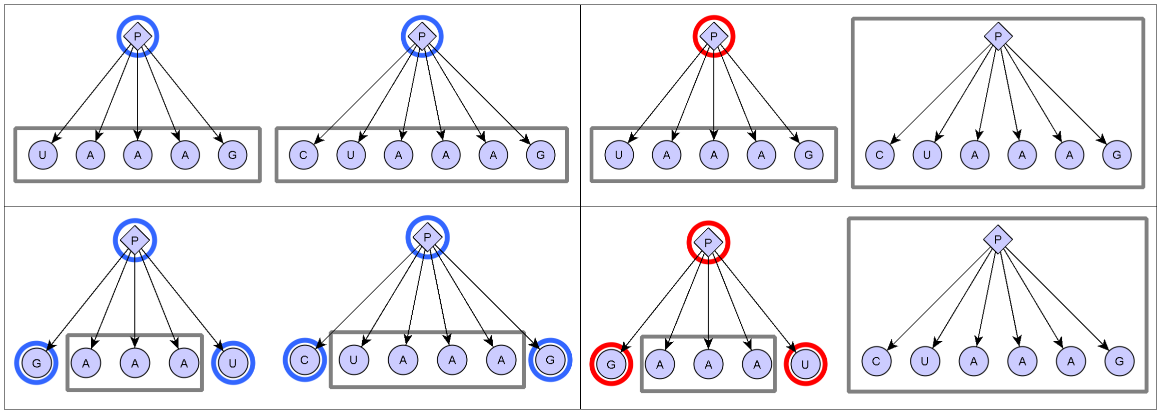


*Table 8: Comparison between 'pair bond' and 'base pair' edit operations. Replacements are blue circles, indels are red circles, and the 'next subforests to be aligned' are in gray boxes. For simplicity, I have not drawn the 'splits' for indel operations. Top left: pair bond replacement doesn't guarantee that the paired bases will actually be aligned. It is likely that the "UAAAG"s will align and the paired "G" will be gapped. Bottom left: base pair replacement solves the problem. Top right: pair bond deletion allows the removal of bonds. Bottom right: base pair indels allow the simultaneous removal of two paired bases.*

Clearly, a base pair replacement can be build from a pair bond replacement and two base replacements, and similarly for indels. However, there are three reasons why we should use 'base pair' operations. First, both pair bond and base pair indels correspond to biologically common mutations,[22] so allowing both operations improves the quality of alignments. Second, base pair replacements allow us to use biologically motivated scoring functions such as the RIBOSUM matrices of (Klein and Eddy, 2003). Third, we should prefer base pair replacement because pair bond replacement leads to structurally invalid alignments. When we replace an individual pair node, we ought to guarantee that its paired bases will also be aligned. Otherwise, we might actually alter one of the aligned structures, shifting a bond so that it lies between a real nucleotide and an artificial 'gap' entity (as in the table above). In this thesis, I won't use pair bond replacement.

Hochsmann *et al.* appear to have forsworn pair bond replacement as well; the `RNAForester` webserver seems to use base pair replacement rather than pair bond replacement (data not shown). Since the algorithm of (Hochsmann *et al.*, 2003) aligns each pair of closed subforests, base pair replacement is easy to implement. Consider a replacement of the pair node $f_i$ such that $pair(f_i) = (f_{5'},f_{3'})$.[23] Recall that in pair bond replacement, we applied a scoring function defined over individual nodes, $score(f_i,g_k)$. We then had to optimize over alignments of the subforest $(f_{5'},f_{3'})$. For base pair replacement, we instead use a scoring function defined over base pairs, $pairscore((f_i,f_{5'},f_{3'}),(g_k,g_{5'},g_{3'}))$. Since the scoring function takes the paired bases into account, we have optimize over subalignments 'inside' them, i.e. alignments of $(next(f_{5'}),prev(f_{3'}))$. We could implement base pair indels in a similar manner - charge a penalty $pairscore((f_i,f_{5'},f_{3'}),(-,-,-))$, then optimize over alignments of $(next(f_{5'}),prev(f_{3'}))$.

22 As discussed in section 3.2.1.

23 I use $(f_{5'},f_{3'})$ to indicate that these are the 5'-most and 3'-most bases of the pair.

We can also implement base pair replacement/indels for the more efficient algorithms of (Jiang *et al.*, 1995; Jansson *et al.*, 2004). In those algorithms we don't consider nonsuffix-nonsuffix alignments; we can get away with it because the loop defined by a base pair, $(f_{5'},f_{3'})$, is always a suffix. However, the loop resulting from a base pair replacement or indel, $(next(f_{5'}),prev(f_{3'}))$, is not a suffix: $prev(f_{3'})$ has $f_{3'}$ as a *next* node. To solve this problem, we just treat all subforests $(f_i,prev(f_{3'}))$ as if they were suffixes. This essentially doubles the number of suffixes in the alignment, but the big-O complexity remains the same.

To be more formal, we have divided the the recurrences for replacements and indels into separate versions for each edit operation:

```
$ALIGN((f_i,f_j),(g_k,g_l))$, base replacement =
    $score(f_i,g_k)$
    $+ ALIGN((next(f_i),f_j), (next(g_k),g_l))$

$ALIGN((f_i,f_j),(g_k,g_l))$, base insertion =
    $score(-,g_k)$
    $+ ALIGN((f_i,f_j), (next(g_k),g_l))$

$ALIGN((f_i,f_j),(g_k,g_l))$, base pair replacement where $pair(f_i)$ is $(f_{5'},f_{3'})$ and $pair(g_k)$ is $(g_{5'},g_{3'})$ =
    $pairscore((f_i,f_{5'},f_{3'}),(g_k,g_{5'},g_{3'}))$
    $+ ALIGN((next(f_{5'}),prev(f_{3'})),(next(g_{5'}),prev(g_{3'})))$
    $+ ALIGN((next(f_i),f_j), (next(g_k),g_l))$

$ALIGN((f_i,f_j),(g_k,g_l))$, base pair insertion where $pair(g_k)$ is $(g_{5'},g_{3'})$ =
    $pairscore((-,-,-),(g_k,g_{5'},g_{3'}))$
    + max over $f_{split}$ in $(f_i,f_j)$ {
        $ALIGN((f_i,f_{split}), (next(g_{5'}),prev(g_{3'})))$
        $+ ALIGN((next(f_{split}),f_j), (next(g_k),g_l))$
    }

$ALIGN((f_i,f_j),(g_k,g_l))$, pair bond insertion where $pair(g_k)$ is $(g_{5'},g_{3'})$ =
    $score(-,g_k)$
    + max over $f_{split}$ in $(f_i,f_j)$ {
        $ALIGN((f_i,f_{split}), (g_{5'},g_{3'}))$
        $+ ALIGN((next(f_{split}),f_j), (next(g_k),g_l))$
    }
```

As before, deletions are just mirrors of insertions and when we maximize over splits, we also maximize over the empty splits $\varnothing$ / $(f_i,f_j)$ and $(f_i,f_j)$ / $\varnothing$.

Now I will modify these recurrence relations to handle forest-RNAPOAs. The modification is mostly straightforward. The only difference between ordered forests and forest-RNAPOAs is that each node can have multiple *next*, *prev*, and *pair* edges. Therefore, wherever a forest alignment computes a value using *next* or *prev*, a forest-RNAPOA alignment optimizes over all possible combinations of *next*s and *prev*s. This creates the same two dangers - invalid spans and wasteful subalignments - that we faced during string-RNAPOA alignment. We'll use the same tricks to solve them: we'll use a partial-ordering matrix $F_{\leq}$ to check span validity, and we'll use a function $ALIGN_{ignore\_end}$ to store certain subalignments efficiently.

To put it explicitly, the recurrences for forest-RNAPOA alignment are:

```
$ALIGN((f_i,f_j),(g_k,g_l))$, base replacement =
        $score(f_i,g_k)$
        + max over $f_{next}$ in next($f_i$), $g_{next}$ in $next(g_k)$ {
                $ALIGN((f_{next},f_j), (g_{next},g_l))$
        }

$ALIGN((f_i,f_j),(g_k,g_l))$, base insertion =
        $score(-,g_k)$
        + max over $g_{next}$ in $next(g_k)$ {
                $ALIGN((f_i,f_j), (g_{next},g_l))$
        }

$ALIGN((f_i,f_j),(g_k,g_l))$, base pair replacement =
        max over $(f_{5'},f_{3'})$ in $pair(f_i)$, $(g_{5'},g_{3'})$ in $pair(g_k)$ {
                $pairscore((f_i,f_{5'},f_{3'}),(g_k,g_{5'},g_{3'}))$
                + max over $f_{next}$ in $next(f_{5'})$, $g_{next}$ in $next(g_{5'})$ {
                        $ALIGN_{ignore\_end}((f_{next}, f_{3'}), (g_{next}, g_{3'}))$
                }
        }
        + max over $f_{next}$ in next($f_i$), $g_{next}$ in $next(g_k)$ {
                $ALIGN((f_{next},f_j), (g_{next},g_l))$
        }

$ALIGN((f_i, f_j),(g_k, g_l))$, base pair insertion =
        max over $(g_{5'}, g_{3'})$ in $pair(g_k)$ {
                $pairscore((-,-,-),(g_k, g_{5'}, g_{3'}))$
                + max over $f_{split}$ in $(f_i, f_j)$ {
                        max over $g_{next}$ in $next(g_{5'})$ {
                                $ALIGN_{ignore\_end}((f_i, f_{split}), (g_{next}, g_{3'}))$
                        }
                        + max over $g_{next}$ in $next(g_k)$ {
                                $ALIGN((f_{split}, f_j), (g_{next}, g_l))$
                }
            }
        }

$ALIGN((f_i,f_j),(g_k,g_l))$, pair bond insertion if $pair(g_k)$ is $(g_{5'},g_{3'})$ =
        max over $(g_{5'}, g_{3'})$ in $pair(g_k)$ {
                $score(-,g_k)$
                + max over $f_{split}$ in $(f_i,f_j)$ {
                        $ALIGN((f_i, f_{split}), (g_{5'}, g_{3'}))$
                        + max over $f_{next}$ in $next(f_{split})$, $g_{next}$ in $next(g_k)$ {
                                $ALIGN((f_{next}, f_j), (g_{next}, g_l))$
                        }
                }
        }
```

Here we use the same three 'tricks' as before. First, $ALIGN_{ignore\_end}((f_i, f_j), (g_k, g_l))$ is just like *ALIGN*, except it ignores the last base in each span. Second, we store and compute a partial-ordering matrix $F_{\leq}$. This allows us to efficiently find all valid split nodes $f_{split}$ and next nodes $f_{next}$ within the span $(f_i,f_j)$. We simply do a depth-first traversal of $F$ starting at $f_i$, only following an edge $next(f_m)$ if $F_{\leq}[m][j] = 1$. Third, as in forest alignment, we have to make sure we consider 'splits' which leave the entire span intact, e.g. $(f_i,f_j)$ / Ø and Ø / $(f_i,f_j)$.

To analyze the complexity of this approach, I will use the following notation:

$|F|$ = the number of nodes in $F$
*Bases*($F$) = the number of base nodes in F
*Pairs*($F$) = the number of pair nodes in F
*Deg*($F$) = the max number of *next* edges from any node in F
*PairDeg*($F$) = the max number of base pairs represented by any pair node in F
*MaxLoop*($F$) = the max number of nodes $f_k$ in any span $(f_i,f_j)$ in $F$, such that
$F_{\leq}[i][k] = 1$ and $F_{\leq}[k][j] = 1$.

What is the space complexity of this algorithm? As in (Jiang *et al.*, 1995), we have to compute suffix-suffix, nonsuffix-suffix, and suffix-nonsuffix alignments. For each end node, we can start a suffix at any of the O(*MaxLoop*($F$)) other nodes in the same loop. Hence there are O(*Pairs*($F$)*MaxLoop*($F$)) 'suffixes' of $F$. On the other hand, any node can end a nonsuffix, so there are O($|F|$*MaxLoop*($F$)) nonsuffixes in $F$. Our modification to use $ALIGN_{ignore\_end}$ essentially forces us to compute each of these alignments twice - once including the end nodes, and once ignoring them[24] - but it doesn't

24 Since some 'ignore end' alignments are very similar to some nonsuffix-suffix alignments, it may be possible to compute their scores without storing them. It wouldn't reduce the big-O complexity, though.

change the big-O complexity. So the number of alignments of each type that we have to compute is as follows:

| | |
|---|---|
| suffix-suffix: | $O(Pairs(F)MaxLoop(F)Pairs(G)MaxLoop(G))$ |
| suffix-nonsuffix: | $O(Pairs(F)MaxLoop(F)\|G\|MaxLoop(G))$ |
| nonsuffix-suffix: | $O(Pairs(G)MaxLoop(G)\|F\|MaxLoop(F))$ |

Also, we have to compute and store the partial-ordering matrices $F_{\leq}$ and $G_{\leq}$, which add time and space complexity $O(|F|^2)$ and $O(|G|^2)$, respectively. The total space complexity is then:

$$\text{space: } O(|F|^2 + |G|^2 + MaxLoop(F)MaxLoop(G)(|F|Pairs(G) + |G|Pairs(F)))$$

What about the time complexity? For each subalignment, we have to consider eight editing operations: base replace, base insert, base delete, base pair replace, base pair insert, base pair delete, pair bond insert, and pair bond delete. If we inspect the recurrences, we see that they have time complexities as follows:

| ***Edit Op.*** | ***Edit Operation Time Complexity*** |
|---|---|
| base replace | $O(Deg(F)Deg(G))$ |
| base insert | $O(Deg(G))$ |
| base delete | $O(Deg(F))$ |
| base pair replace | $O(PairDeg(F)PairDeg(G)Deg(F)Deg(G))$ |
| base pair insert | $O(MaxLoop(F)PairDeg(G)Deg(G))$ |
| base pair delete | $O(MaxLoop(G)PairDeg(F)Deg(F))$ |
| pair bond insert | $O(MaxLoop(F)PairDeg(G)Deg(F)Deg(G))$ |
| pair bond delete | $O(MaxLoop(G)PairDeg(F)Deg(G)Deg(F))$ |

Base pair operations dominate the running time, so the total time complexity is:

$$\begin{aligned}\text{time: } O(&|F|^2 + |G|^2 + \\ &MaxLoop(F)MaxLoop(G)(|F|Pairs(G) + |G|Pairs(F)) \\ &* \ Deg(F)Deg(G) * (PairDeg(F)PairDeg(G) \\ &\qquad + MaxLoop(F)PairDeg(G) \\ &\qquad + MaxLoop(G)PairDeg(F)))\end{aligned}$$

Most 'real world' situations are much simpler to analyze. For individual RNA structures, $PairDeg(F) = Deg(F) = 1$ and $MaxLoop(F)Pairs(F) = O(|F|)$. If we do a little algebra, the complexities reduce to $O(|F|^2 + |G|^2 + |F||G|(MaxLoop(F) + MaxLoop(G)))$

space and O($|F|^2 + |G|^2 + |F||G|(MaxLoop(F) + MaxLoop(G))^2$) time. This is essentially the same as the algorithms of (Jiang *et al.* 1995; Jansson *et al.*, 2004). For multiple RNA structures, we can make the same argument that we did for string-RNAPOAs. So long as they are reasonably similar, *Deg*(*F*) and *PairDeg*(*F*) will be small constants, so alignment time shouldn't increase very quickly with the number of aligned structures.

A major drawback of this type of forest-RNAPOA alignment is that it enforces the same nesting constraint as ordinary forest alignments. To visit each base node of an ordered forest, we have to pass through each pair node. Hence an alignment's 'threading' of *F* has to visit every pair $f_p$ in *F* (even if the pair is indel'd anyway). However, if we look at the recurrence relations for forest alignment, any alignment visitng $f_p$ gets divided into two subalignments - one starting at *next*($f_p$) and another enclosed by *pair*($f_p$). Likewise, any alignment visitng $g_p$ is divided in the same way. But since all alignments have to visit both $f_p$ and $g_p$, our choice of alignments is limited. If we align any base in *pair*($f_p$) with any base in *pair*($g_p$), then no base in *pair*($f_p$) can be aligned with *next*($g_p$) and no base in *pair*($g_p$) can be aligned with *next*($f_p$). In other words, the alignment can't make $f_p$ and $g_p$ cross.

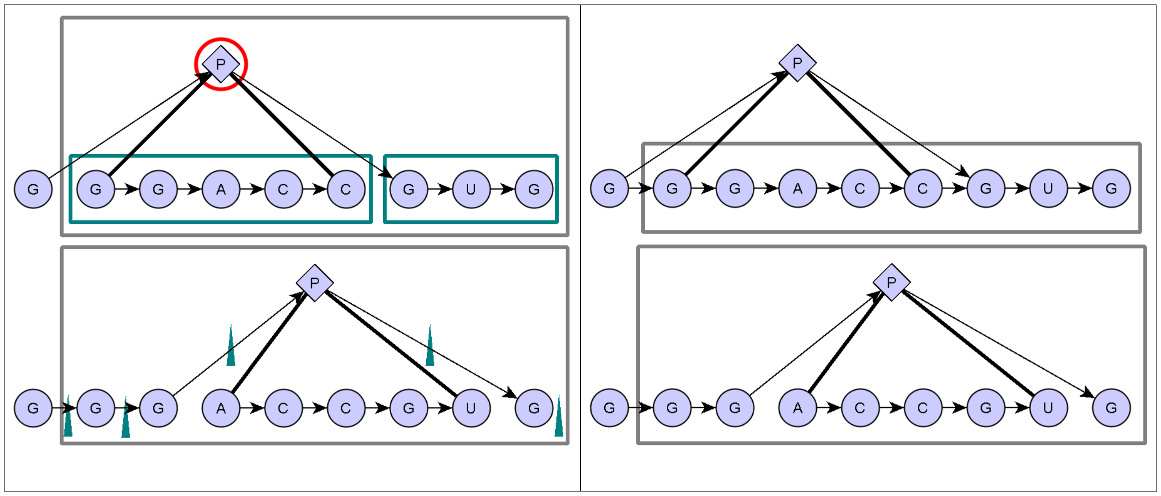


*Table 9: Forest-RNAPOAs F (top) and G (bottom) and their adventures with crossing base pairs. Left: with ordinary forests, an alignment's threading must visit these spans (gray boxes). If we delete the pair node (red circle), we divide F into two parts (blue boxes). None of the split points in G (blue triangles) will let us match all of the identical nucleotides. Right: if F is a 'stringlike' RNAPOA, we can choose an alternate path from the leftmost "G" base that doesn't visit the pair node. The alignment is no longer constrained to 'nest'.*

We can get around this problem by using 'stringlike' RNAPOAs. A stringlike forest-RNAPOA, just like an string-RNAPOA, treats each base pair as an optional, alternate path. Consider an optimal alignment *A* of a stringlike RNAPOA *F* and any arbitrary forest-RNAPOA *G*. What if the *A* makes a pair node $f_p$ cross a pair node $g_p$? That is, what if *pair*($f_p$) = ($f_{5'}$,$f_{3'}$) and *pair*($g_p$) = ($g_{5'}$,$g_{3'}$), but *A* aligns some base nodes in

($f_{5'}$,$f_{3'}$) with base nodes that aren't in ($g_{5'}$,$g_{3'}$)? The worst case is that $G$ is an ordinary forest, so the alignment's threading of $G$ has to visit $g_p$ (if only to indel it). However, the threading of $F$ doesn't have to visit $f_p$ , since $f_p$ is on an optional, alternate path. Hence we can still generate $A$ as the optimal alignment.

This freedom comes at the cost of efficiency. If $F$ is stringlike, then $MaxLoop(F) = |F|$, so a pairwise alignment requires $O(|G|^2 + |F||G|(|F| + MaxLoop(G)))$ space and $O(|G|^2 + |F||G|(|F| + MaxLoop(G))^2)$ time. It is straightforward to interconvert ordinary and stringlike RNAPOAs, so we can always choose the smaller structure to be stringlike.

A final note: it is very inefficient to align base pairing probability matrices using forest-RNAPOAs. For a base-pairing probability matrix, $|F|$, $Pairs(F)$, $MaxLoop(F) = O(Bases(F)^2)$, and $PairDeg(F) = Bases(F)$. e.g. when $F$ and $G$ have comparable sizes, the algorithm would require $O(Bases(F)^8)$ space and $O(Bases(F)^{11})$ time.

## *3.3 Approximate alignment of RNAPOAs with tertiary structure*

The alignment algorithms that I developed in the previous section do not align two entire RNAPOAs to each other. They align a single 'threading' of $S$ onto $T$. When I say 'threading', I simply mean the nodes of $S$ and $T$ that are aligned with each other. RNAPOAs, like partial-order sequence alignments, only generate an optimal threading of any one sequence of S with any one sequence of $T$. Also, like arc-annotated structure alignments, they only generate an optimal threading of any one secondary structure of S with any one secondary structure of $T$ - no 'alternate' or crossing base pairs are allowed.

In general, we'd like to do more - we want to align tertiary structure as well as secondary structure. In addition, the nature of partial-order alignments gives us another reason to go beyond a single threading. To see why, let us first consider the simpler case of PO-MSAs.

Imagine two identical PO-MSAs $S$ and $T$. Since they are identical, $S$ and $T$ have the same alternate paths. When we align $S$ and $T$, we find the best threading of a path in $S$ to a path in $T$. Then we merge the aligned nodes to form a new PO-MSA $A$. What happens to the alternate paths that weren't threaded? We don't merge them, so they remain in $A$ as duplicate paths. In theory, this duplication increases the time and space requirements of future alignments and decreases their quality. Future alignments could add different replacements/indels to each duplicate path, but the two ought to be identical. Of course, these statements about 'duplicates' also apply to any alternate paths, not just duplicates; even if they aren't on the optimal threading, we still ought to align them.

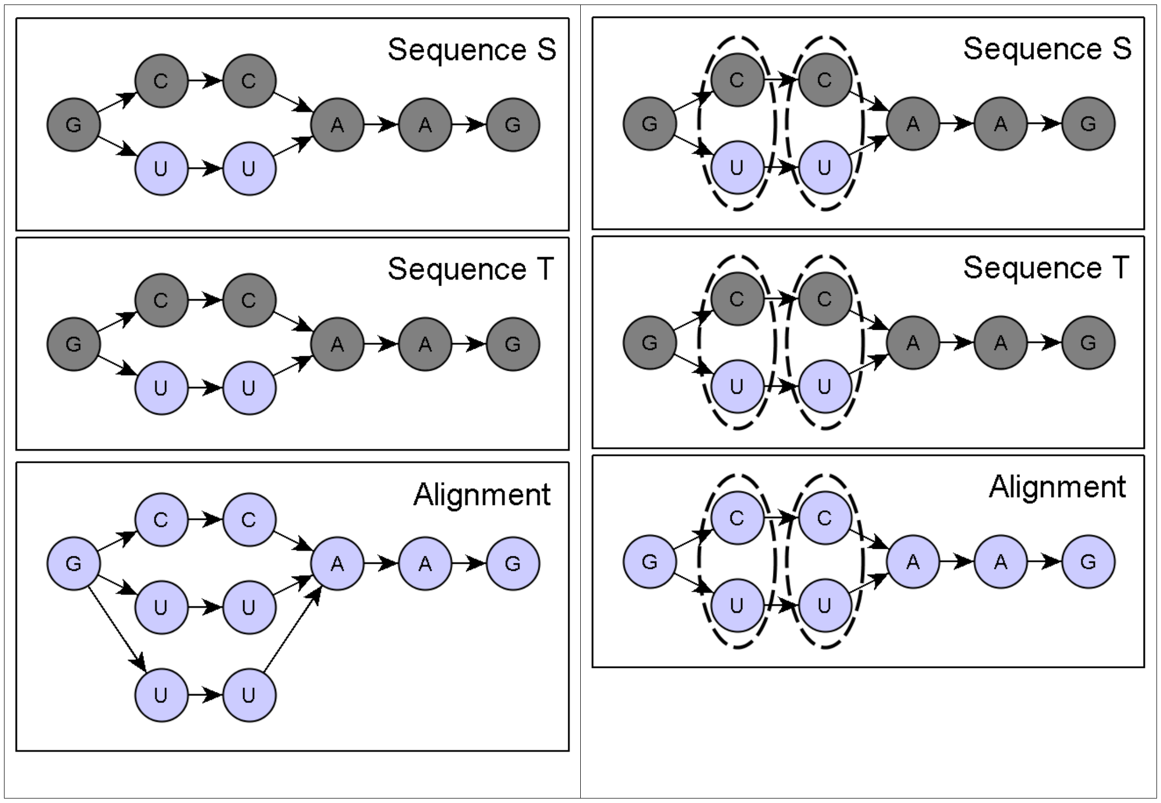


*Table 10: A hypothetical peril of alternate paths in PO-MSAs. Left: two identical PO-MSAs and their alignment. Aligned nodes are colored gray. The alignment is a 'threading' of a single path in S onto a single path in T. If S and T have alternate paths, those paths aren't merged. Right: in practice, indel costs are much higher than mismatch costs, so 'alternate paths' are merely mismatched (dashed circles). Aggressive merging of aligned, mismatched nodes solves the problem.*

We would expect this to be a big problem for PO-MSAs - unless the guide tree is very good, alignments should accumulate duplicate paths. In practice, these effects are small. It is difficult to generate this kind of alternate path, since scoring functions typically give higher penalties to sequence indels than to subsitution mutations.

RNAPOAs can't escape the problem so easily. After all, a major purpose of RNA structure alignment is to take structural information into account when deciding where to place indels. If we only consider one 'threaded' structure of many, we are throwing away a lot of information. In the most extreme case - RNAPOAs that represent full base pairing probability matrices - we only use a small fraction of the total structural information.[25] This is an especially dangerous issue for forest-RNAPOAs. Since they represent base pairs explicitly, any base pairs that cross the 'threaded' structure will not be aligned and could easily become duplicated. For example, if a moderate-sized substructure of an RNA has two common variations, and the variations appear in several different phylogenetic

25 The same criticism applies to the total order approach of (Hofacker *et al.*, 2004).

groups, a forest-RNAPOA would quickly accumulate duplicate paths containing many extra pair nodes.

The main challenge here is to take crossing base pairs into account during alignment; if we can do that, we can handle tertiary structure and avoid duplicate paths. For string-RNAPOAs, we can solve this problem with a variation of the 'constrained alignments' outlined by (Wang and Zhang, 2001).[26] The basic idea is as follows. In the first pass, we align $S$ and $T$ to produce the alignment $A_1$. Then we generate constraints to force future passes to respect the aligned base pairs in $A_1$. Finally, we remove those base pairs from $S$ and $T$ (so that we can cross them), and re-align $S$ and $T$ according to the constraints. We keep repeating this process, adding more constraints and producing more refined alignments $A_2$, $A_3$, $A_4$, etc. We stop when $A_p = A_{p-1}$, i.e. when the next alignment doesn't include any new base pairs.

Now for a more detailed explanation. Let *aligned_base*($s_i$) = $t_k$ if $s_i$ is constrained to align with $t_k$, and *aligned_base*($s_i$) = empty otherwise. Let *ignored_pairs* be a set of base pairs that have already been aligned in some previous alignment pass $A_p$. After we generate an alignment, for each pair of aligned base pairs ($s_i$,$s_m$) and ($t_k$,$t_n$), we set:

*aligned_base*($s_i$) = $t_k$
*aligned_base*($s_m$) = $t_n$
*aligned_base*($t_k$) = $s_i$
*aligned_base*($t_n$) = $s_m$
*ignored_pairs* = *ignored_pairs* U {($s_i$,$s_m$), ($t_k$,$t_n$)}

Now that we have our constraints, we apply them to *ALIGN*(($s_i$,$s_j$), ($t_k$,$t_n$)) as follows. First, we force the alignment to respect previously-aligned paired bases. If *aligned_base*($s_i$) = $t_k$ and *aligned_base*($t_k$) = $s_i$, then we don't allow indels. If we are forced to accept an indel - e.g. if *aligned_base*($s_i$) is not empty but *aligned_base*($t_k$) = empty - then we give it a score of $(-\infty)$. If *aligned_base*($s_i$) $\neq t_k$ and is not empty, or *aligned_base*($t_k$) $\neq s_i$ and is not empty, then we don't allow base replacements. Second, we don't consider base pairs that were in previous alignments. When we optimize over all base pairs in *pair*($s_i$), we ignore any pairs ($s_i$,$s_m$) that are in *ignored_pairs*. We also ignore any pairs ($t_k$,$t_n$) that are in *ignored_pairs*. That's all!

The constraints are easy to implement without changing the time or space complexity of alignment. Actually, the constraints make alignments considerably more efficient. We don't need to compute as many subalignments, since we don't have to align pair-suffix spans if we are ignoring all of the pairs that depend on that span. Nevertheless, the worst-case complexity is the pairwise complexity times the number of passes: about O(*Bases*(*S*)*Bases*(*T*)*min*(*Bases*(S),*Pairs*(*S*))*min*(*Bases*(*T*),*Pairs*(*T*))(*#passes)*). If we are dealing with tertiary structures or string-RNAPOAs with alternate paths, this procedure should stop aligning new base pairs after a small number of passes. Alignments with many more pair nodes than bases could potentially take much longer, since each pass

26 As discussed near the end of section 2.5.1.

adds at most O(*Bases*(*S*)) base pairs to *ignored_pairs*. In such cases we might impose an upper limit to the number of passes.

The method of (Wang and Zhang, 2001) preserved a total ordering, so each base in the alignment had only one structural feature. In contrast, this partial-order method can actually align multiple base pairs for the same base. For example, it can align $(s_i,s_m)$ and $(t_k,t_n)$ in the first pass, and then $(s_i,s_a)$ and $(t_k,t_b)$ in the second pass. Thus the resulting RNAPOA is not merely an alignment of two consensus structures or structural threadings; it is an alignment of the entire set of alternate structures shared between two ensembles of RNAs. The alignment is built greedily from a series of threadings of secondary structure, so it may not be globally optimal. However, much of the similarity between homologous structure ensembles can be captured by a single secondary structure threading (e.g. see the results of (Hofacker *et al.*, 2004)), so for most practical cases this greedy method should give good results.

We can use the same basic strategy for forest-RNAPOAs. The *aligned_base* constraint can be implemented just as for string-RNAPOAs. The *ignored_pairs* constraint is a bit trickier; we can't just ignore pair nodes, since we have to visit them when we traverse the graph. Instead, whenever we compute $ALIGN((f_i,f_j), (g_k, g_l))$, we force a pair bond insertion if $g_k$ is in *ignored_pairs* and force a pair bond deletion if $f_i$ is in *ignored_pairs*. This way we basically skip over all pair nodes that have already been aligned. Again, the algorithm's complexity is the same; the total cost is just the pairwise cost times the number of passes. Note that this method is mostly useful for alignments that include a stringlike forest-RNAPOA; otherwise, the forest representation stops us from considering crossing base pairs.

## 3.4 Approaches to multiple alignment

The main focus of this thesis is the RNAPOA representation of multiple alignments and algorithms for aligning two RNAPOAs. With these in hand, there are many methods we could use to build multiple alignments. Here I will review three such methods and briefly discuss the advantages and disadvantages of each. Current RNA multiple structural alignment programs use techniques originally developed for sequence alignment. It may be that RNAs have certain properties that would benefit from specialized alignment-building schemes. However, the methods that I will discuss here are relatively straightforward and can be used with any type of 'alignment of alignments'.

The simplest way to build a multiple alignment is to merge single structures into an alignment one-by-one in an arbitrary order. I will call this the *linear* scheme, because it requires O(*k*) total alignments for *k* structures. With traditional, total-order sequence alignments, this approach tends to produce bad alignments. On the other hand, (Lee *et al.*, 2002) used this simple scheme with partial order sequence alignments and got fairly good results.

A more traditional way to build a multiple alignment is to construct a 'guide tree' that determines the order in which we merge a series of intermediate alignments. I will call this the *tree-guided* scheme. We can choose a guide tree by applying a phylogenetic tree reconstruction algorithm (e.g. (Saitou and Nei, 1987)) to a matrix of pairwise distances between the input structures. To generate the distance matrix, we need to perform O($k^2$) pairwise structure alignments. The guide tree will include O($k$) internal nodes, and we have to perform one pairwise alignment for each one.

Two of the programs that I've mentioned in this thesis - `pmmulti` (Hofacker *et al.*, 2004) and `POA2` (Grasso and Lee, 2004) - use this approach. Because it takes so many pairwise alignments to generate the distance matrix, both of these programs provide a way to avoid using the full, optimal alignment algorithm during this step. `pmmulti` has an option to use 'string-like' alignments of base pairing probabilities, and `POA2` uses `BLAST` (Altschul *et al.*, 1997), a fast, approximate sequence comparison algorithm. Even these rough guide trees significantly improved the quality of PO-MSAs.

`RNAForester` uses a third approach. One use of multiple alignments is as input to programs for inferring phylogenetic trees. However, (Hochsmann *et al.*, 2004) note that inferred trees are strongly influenced by the alignment's choice of guide tree. In that case, the multiple alignment isn't telling us anything about sequence phylogeny that we didn't already know from the individual pairwise alignments. To avoid this dilemma, `RNAForester` iteratively constructs a 'guide tree' during the course of alignment. I will call this the *tree-building* scheme. The idea is as follows:

1. Compute all O($k^2$) pairwise distances between the $k$ single-structure RNA profiles.
2. Align the two most similar RNA profiles. Merge them into new profile $A$.
3. Compute all O($k$) pairwise distances between $A$ and the other profiles.
4. If more than one profile remains, go to step 2.

So, rather than trusting a guide tree to figure out ahead of time which intermediate profiles will have the smallest distances to each other, `RNAForester` recomputes pairwise distances as it goes. For k structures we need only ($k$-1) pairwise alignments to merge them all. If we compute O($k$) distances after each step, we only do O($k^2$) recomputations in all. Thus all of the additional pairwise alignments double the running time but don't increase the alignment complexity.

Which of these approaches is best for RNAPOAs? It depends on the purpose of the alignment. If speed is an issue, or if there are a large number of structures, the linear method is the only one that will be efficient. The advantages of the partial order formulation should let RNAPOAs give fairly accurate alignments even when the input structures are merged one by one. If we are doing a phylogenetic comparison in which accuracy is paramount, the more rigorous methods are the way to go. Following (Hofacker *et al.*, 2004), we might build a guide tree based on 'string-like' alignments to keep the running time manageable for large sequence lengths.

Unfortunately, although I have implemented the linear method, I have yet to evaluate the other two. Given the improvement that even a rather rough guide tree makes to PO-MSAs, it is likely that RNAPOAs will benefit from the tree-guided and tree-building methods as well.

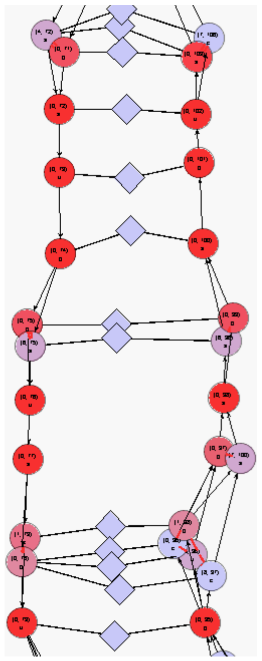
## 4. Experimental Results

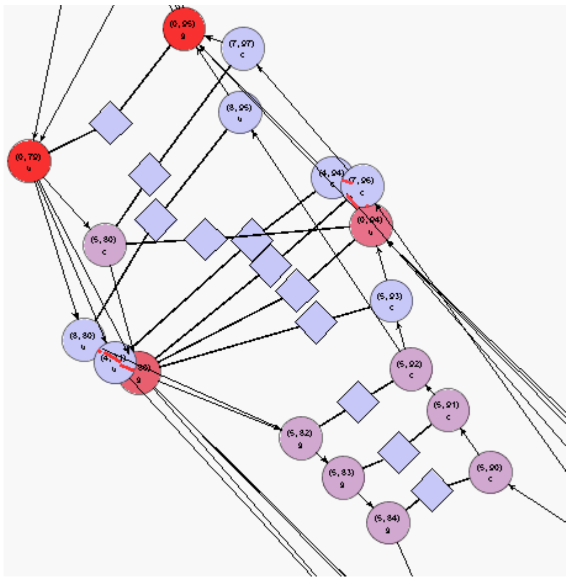
In this section I will present a few preliminary results. Before I begin, I should mention an important caveat. In the realm of sequence alignments, standardized alignment databases such as BaliBase (Bahr *et al.*, 2001) make it easy for researchers to to objectively compare sequence alignment algorithms. However, RNAPOAs are partial-order alignments and are RNA structural alignments. Both of these properties make RNAPOAs difficult to compare with existing methods. First, as (Lee *et al.*, 2002) noted, a PO-MSA represents a very large number of equivalent (but differently scored) total order alignments. Therefore it is something of an apples-to-orangles comparison to evaluate PO-MSAs accoring to row-column criteria. A measure like 'percent conserved residues correctly aligned' essentially compares an alignment to a profile, even if the alignment is better represented by a partial ordering with alternate paths.

Secondly, RNA structure alignments are even more difficult to objectively compare. We might use RNA structural alignments for several tasks - to find similar structures from a database, determine a consensus structure for a phylogenetic group, infer phylogenetic relationships from aligned RNAs, or explore the variation within an ensemble of computationally predicted structures. In each case, we have different criteria for what makes an alignment 'good'. For example, repositories like the RNAse P Database (Brown, 1999) and BRAliBase (Gardner and Giegerich, 2004) have structures for RNAs from many different phylogenetic groups. The base pairs of each structure are generally determined by aligning many sequences at once to consensus structures that have been determined by experimental studies and phylogenetic analysis. In one sense this is the 'gold standard' for alignment programs, since the consensus is our best estimate of the actual structure of the RNAs. However, if most of the RNAs can be fitted to the same consensus structure, then it nullifies the advantages of RNAPOAs in capturing variations as well as commonalities.

In any case, the above points are academic for now. I have implemented string-RNAPOA and forest-RNAPOA alignment, but the algorithms are still in an experimental state and aren't yet ready for a thorough evaluation. The code is written in python, using the Numeric module for multidimensional arrays and the Psyco module for code optimization. For reference, I can give some wall clock times for string-RNAPOA alignment: it takes 90 seconds to align two 5S RNA structures with about 120 bases each, 5 minutes to align two RNAse Ps with about 270 bases each and a small number of base pairs, and 30 minutes to align two RNAse Ps with about 350 nucleotides each and a larger number of base pairs. I expect that a C++ implementation would be several times faster.

Now I can show some results. The following images are from a multiple alignment of nine bacterial 5S RNA structures obtained from the Comparative RNA Web Site (Cannone *et al.*, 2002). Each sequence is about 120 nucleotides long, and the RNAs have a well-conserved structure. I used the graph editing program `yEd` to render and arrange the resulting RNAPOAs. In each image, circular nodes represent bases and diamond nodes represent base pairs. Bases are colored light blue if they occur in only one structure, but become red in proportion to the number of structures in which they appear. All of the bases at a single position in the alignment - i.e. matched and mismatched bases - are connected by by red dashed lines. These are not always visible. To remedy this, I have moved all aligned base nodes so that they partially overlap, which seems to be a better visual cue.

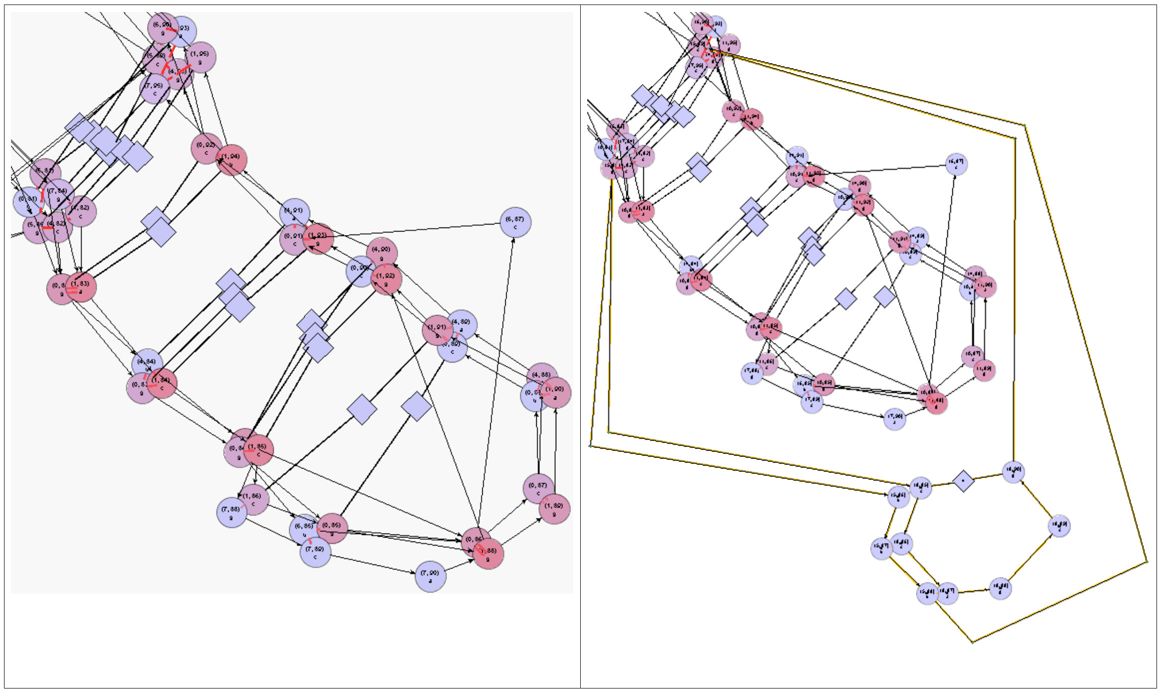




The left image in the table above shows a region with conserved sequence and structure. There are several nodes which have exactly the same nucleotide and base pair in all nine structures. There are a few alignment positions which have alternate base pairs; for example, the position in the middle of the image has two aligned base pairs, one of which is very common (bright red) and the other of which only occurs in a few structures (light pink). One of the base pairs near the bottom of the image is extremely variable - one of its bases has two variants and the other base has all four possible nucleotides.

The right image shows a region that is less well conserved; there are several alternate paths through this part of the RNAPOA. In the center of the image, there are a few adjacent positions whose nucleotides and pairing partners vary. Several of the base pairs cross, representing base pair shifts. In addition, some of the base nodes have *next* edges to several different positions. For example, the fully conserved base node (bright red) in the top left has next edges to two different positions. One *next* is a variable position, with two rare bases (light blue) and one common base (red). The other *next* (purple) is an insertion in front of the variable position. Furthermore, these two *next* positions have highly variable structure; each alternate nucleotide can form several different base pairs. In the bottom right of the image, we can see a high-level variation: three consecutive base pairs (purple) which are present in some, but not all, of the RNAs.

I should note that the alignment quality is rather poor. In particular, the scoring function I'm using is very simple and it better suited to testing toy examples than to alignment of phylogenetically related RNAs. An accurate multiple alignment of these structures would show very little variation. Nevertheless, these RNAPOAs show how partial order alignments can represent complex indels, base pair shifts, and subtle interactions between sequence and structure variations.

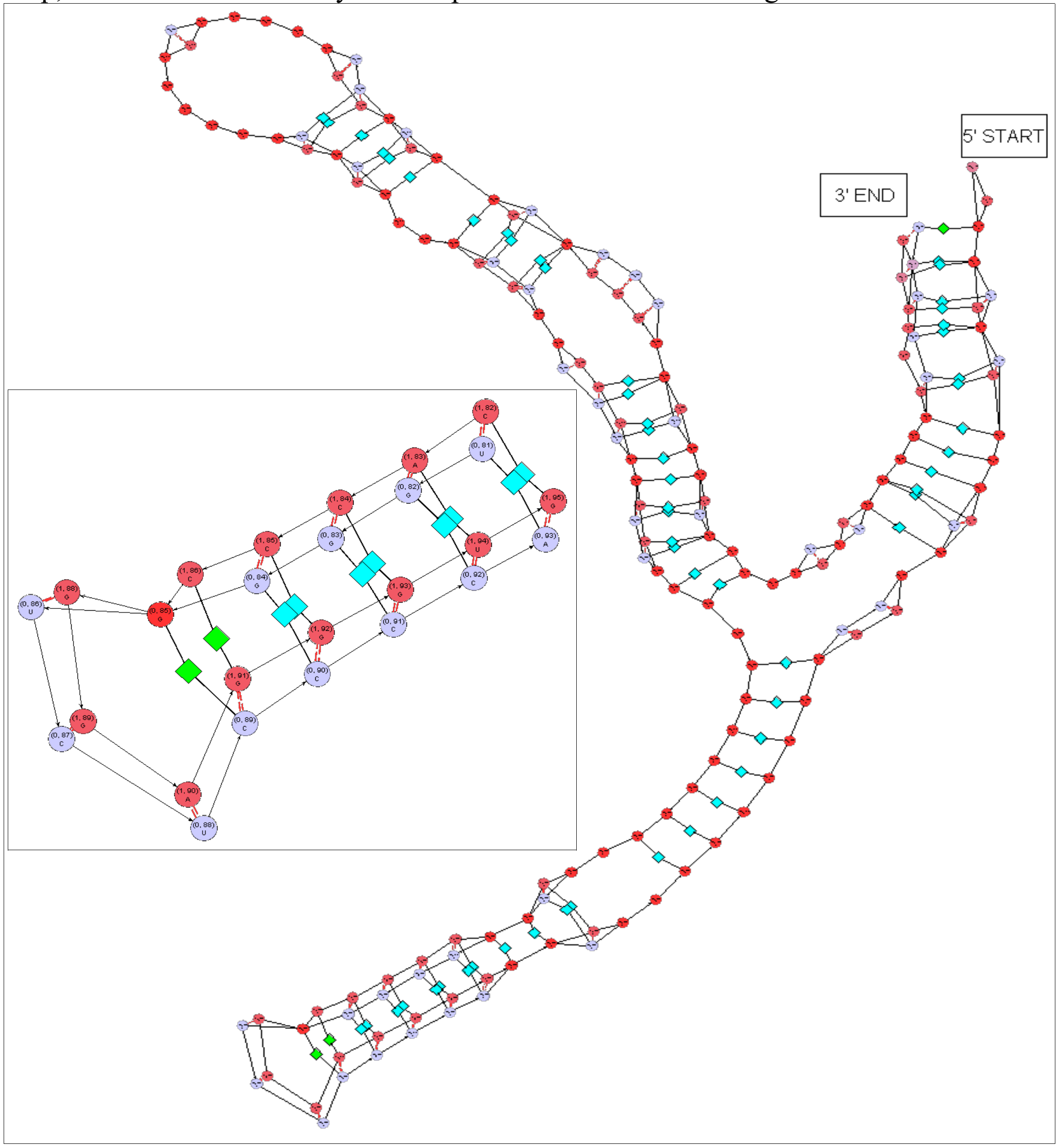


These two images show two different views of a stem and a four-base hairpin loop. The hairpin's structure is fairly well conserved, but the nucleotide content is extremely variable. In particular, consider the paired bases at the upper left corner of the left image. Each of those paired positions has many different nucleotides, and some nucleotides even have more than one pairing partner.[27] The other positions are only slightly less variable in nucleotide content. In contrast, the structure is mostly conserved, but we can still see a few variations. Some positions have mutations that prevent them from forming base pairs. For example, the bottom-most nodes in the image (light blue) don't participate in base pairs, although they are aligned with other base nodes that do

27 In fact, there are more than four base nodes aligned at that position, due to a bug in my code for merging aligned nodes.

form pairs. Also, a small number of structures have base indels near the end of the hairpin - note the crossing *next* edges and the wayward (light blue) base node on the right.

The right image shows a view of the region from farther back. Although the full substructure is present in most of the RNAs, two of them have an alternate, truncated form (highlighted in yellow). One of the RNAs has a base pair and a four-base hairpin loop, while the other has only three unpaired bases and an indel against the rest.



The image above shows an accurate alignment of four of the 5S RNAs. Base pairs present in all structures are colored bright blue, and variable base pairs are colored green. Their strongly conserved structure makes it a rather boring case, but we can see a few places where the partial ordering formulation is useful. The RNAPOA represents mutated regions as alternate paths rather than as a profile, and shows how sequence mutations present in one RNA create a new base pair at the 3' end of the structure and induce a base pair shift near a hairpin loop. One RNA has many substitution mutations, forming an alternate path through the graph that induces only a small change in the structure (see inset).

The timing statistics of this alignment show some of the algorithmic properties of RNAPOAs. In a messy graph structure like the one depicted above, we would expect *BaseDeg* and *PairDeg* to increase significantly, and hence force us to maximize over many more possibilities at each step. Even so, the initial pairwise alignment processed about 50,000 subalignments per second, and the final alignment of the RNAPOA to its last structure processed about 45,000 per second. At least for the linear alignment scheme, those pesky degree terms appear to be small in magnitude and slow in growth, as expected.

However, the total alignment time was not as stable. For the initial pairwise alignment, the first structure had 1968 suffixes/pair-suffixes and the second had 1881, so the algorithm had to compute a total of 3.7 million subalignments - about 70 seconds. However, the RNAPOA accumulated many additional nodes during the course of alignment. By the last step, it had 11231 suffixes/pair-suffixes, whereas the last individual structure had only 1758. Hence the final merging required 20 million subalignments - about 8 minutes.

I should caution that this is an isolated data point measured for an algorithm in development. These results may be artifacts of the simple scoring function I used, or even bugs in my implementation. In addition, I haven't yet implemented the $\mathit{ALIGN}_{\mathit{ignore\_end}}$ trick, which should greatly reduce the effect of node accumulation on the number of subalignments that have to be computed. I will have to test the algorithm more extensively to give a full evaluation of its properties 'in practice'.

# 5. Discussion

## *5.1 Contributions*

Individual RNAs must obey many structural constraints: each position can have only one nucleotide; each nucleotide can have at most one predecessor, one successor, and one pairing partner; if the RNA has only secondary structure, its base pairs do not cross. However, an *alignment* of RNA structures should not obey the same constraints.

Early approaches to sequence analysis made no distinction between a sequence and an alignment of sequences. An alignment was simply a single sequence with a consensus nucleotide at each position. Profile-based approaches relax the 'one nucleotide per position' constraint, which greatly improves alignment quality. However, this requires the use of artificial 'gap' entities. Partial-order alignments relax the 'one predecessor, one successor' constraint and achieve an even more natural representation.[28]

In the same vein, current RNA structure comparison methods do not make a clean separation between individual sequences, individual structures, and alignments. In this thesis, I have addressed this problem with a partial order formulation of RNA structure alignment. An RNAPOA faithfully represents an set of individual structures without splicing in 'gaps' or enforcing consensus features. In particular, RNAPOAs can handle base pair shifting, a common structural mutation that cannot be represented under a total ordering. This natural representation has the potential to considerably improve the quality of alignments. Each aligned secondary structure is a path through an RNAPOA, and when we find an optimal threading of two RNAPOAs, we find an optimal threading between any two of their paths. In addition, RNAPOAs support an iterative method for approximate alignment of 'alternate paths'. Not only does this allow for alignments of tertiary structure, but it also minimizes a subtle source of error in current partial-order alignment methods.

The table below summarizes the algorithms presented in this thesis. They are based on existing structure alignment algorithms, so they generally have similar complexity. In some cases they require additional time/space to relax the structural constraints of a total ordering. I present algorithms for two types of alignments: string-RNAPOAs and forest-RNAPOA. Forest-RNAPOAs are good when the structures are generally described by a common nesting, since *MaxLoop*(*F*) is much smaller than *Pairs*(*F*). On the other hand, string-RNAPOAs are best when the alignment contains many crossing or alternate base pairs, since we don't have to visit pair nodes during alignment. Both string-RNAPOAs and forest-RNAPOAs contain essentially the same information, so it is straightforward to interconvert them when one or the other representation is advantageous. Indeed, although I have chosen to treat them as separate types of graph, we could likely unify them into a single type of graph structure which supports both string-like and forest-like algorithms.

28 The method of (Raphael *et al.*, 2004), further relaxes these constraints to allow cyclic graphs, which represent shuffled/repeated sequence elements.

| ***Representation and algorithm*** | ***Edit Operations*** | ***Space Complexity*** | ***Time Complexity*** | ***Notes*** |
|---|---|---|---|---|
| `RSMatch` tree<br>(Liu *et al.*, 2005) | unpaired base {replace, indel}<br>base pair replace<br>base-pair-and-substructure indel | O($Bases(S)^2$) | O($Bases(S)^2$) | Very strict matching. Deletes entire substructures rather than aligning unpaired and paired bases. |
| RNA profile (forest)<br>(Hochsmann *et al.*, 2004) | base {replace, indel}<br>base pair replace<br>pair bond indel | O($\|F\|^2 MaxLoop(F)$ ) | O($\|F\|^2 MaxLoop(F)^2$ ) | Enforces a consensus nesting; i.e. cannot represent crossing pairs. |
| arc-annotated alignment<br>(Wang and Zhang, 2004) | unpaired base {replace, indel}<br>base pair {replace, indel} | O($Bases(S)^2 Pairs(S)^2$ ) | O($Bases(S)^2 Pairs(S)^2$ ) | Enforces a consensus structure; cannot align unpaired and paired bases. Approx. tertiary structure alignment. |
| base-pairing probability matrix<br>(Hofacker *et al.*, 2004) | base {replace, indel}<br>base pair replace | O($Bases(S)^4$)<br>constrained: O($Bases(S)^3$) | O($Bases(S)^6$)<br>constrained: O($Bases(S)^4$) | Enforces consensus pairing probabilities. |
| ***Introduced in this thesis:*** | | | | |
| string-RNAPOA (1) | base {replace, indel}<br>base pair replace | O($Bases(S)^2 Pairs(S)^2$) | O($Bases(S)^2 Pairs(S)^2$) | Partial-order. Approx. tertiary structure alignment. |
| string-RNAPOA (2)<br>(allows base pair indel) | base {replace, indel}<br>base pair {replace, indel} | O($Bases(S)^4$) | O($Bases(S)^5$) | Partial-order. Approx. tertiary structure alignment. |
| string-RNAPOA (3)<br>(for pairing prob. matrix) | base {replace, indel}<br>base pair {replace, indel} | O($Bases(S)^4$) | O($Bases(S)^6$) | Same as (1); Even when $Pairs(S)$ = O($Bases(S)^2$), the complexity doesn't grow beyond this. |
| forest-RNAPOA (1) | base {replace, indel}<br>base pair {replace, indel}<br>pair bond indel | O($\|F\|^2 MaxLoop(F)$) | O($\|F\|^2 MaxLoop(F)^2$) | Partial-order, but enforces a consensus nesting. |
| forest-RNAPOA (2)<br>(for crossing base pairs) | base {replace, indel}<br>base pair {replace, indel}<br>pair bond indel | O($\|F\|^3$) | O($\|F\|^4$) | Partial-order. Approx. tertiary structure alignment. |

*Table 1: Summary of the algorithms in this thesis, in comparison to other multiple alignment methods. Edit operations include edits restricted to certain classes of nucleotides (e.g. unpaired base replacement), to paired bases (e.g. base pair indel), or just to the bond between two bases (e.g. pair bond indel). Algorithm space/time requirements are given in terms of pairwise alignments between two structures with similar sizes. When we consider multiple alignments, complexity grows in a small but complicated way related to the number of gaps or alternate paths in an alignment. Complexity notation: Bases(S) = number of nucleotides in S, Pairs(S) = number of base pairs in S, |F| = number of nodes in a graph = (Bases(F) + Pairs(F)), MaxLoop(F) = number of bases in the largest loop in F.*

### 5.2 Future Work

RNAPOAs are rather slow to compute, for two main reasons. First, RNAPOAs respect sequence and structure information, but they don't use them as hard constraints. If we permit structurally invalid alignments, we can just use a sequence-based algorithm with quadratic complexity. If we force all aligned RNAs to have similar types of structures, we can again achieve quadratic or near-quadratic complexity (Hochsmann *et al.*, 2004; Liu *et al.*, 2005). On the other hand, RNAPOAs allow "sequence *or* structure" alignments, so they have to optimize over a much larger search space. For example, a pairwise sequence alignment of 120 nt 5S RNAs considers about 15,000 subalignments, but a string-RNAPOA alignment considers nearly 4 million.

We could remedy this problem by imposing some reasonable restriction on the search space. For example, the algorithm of (Hofacker *et al.*, 2004) uses an approximation parameter Δ to reduce alignment time compleixty from sextic to quartic. The idea is to ignore subalignments in which one span has Δ more nucleotides than the other; instead of aligning each span in *S* with each all $|T|^2$ spans in *T*, we only try $\Delta^2$ spans in *T*. In other words, *S* and *T* can only be 'out of register' by Δ positions. (Bafna and Zhang, 2004) and (Yang and Blanchette, 2004) use a somewhat similar approach, in which they only search for alignments that are within Δ of an approximate, sequence-based alignment. In general, most approximation methods for total-order alignments should also work for partial-order alignments.

A second reason why RNAPOAs are slow is that RNAs with conserved structure may have highly variable sequences. In that case, a partial order alignment pays a high cost for preserving every sequence/structure variation. Progressive alignment is an approximation heuristic anyway, so it may not be worth it to pay this cost if it gives only a marginal improvement in alignment accuracy. (Grasso and Lee, 2004) suggest that a reasonable tradeoff is to merge mismatched bases into profile nodes. This reduces the number of nodes while still preserving the main advantages of the partial order formulation. A similar approach would produce even greater speedups for RNAPOAs, since each extra node/edge forces us to consider O(*Bases*(*S*)) additional subalignments.

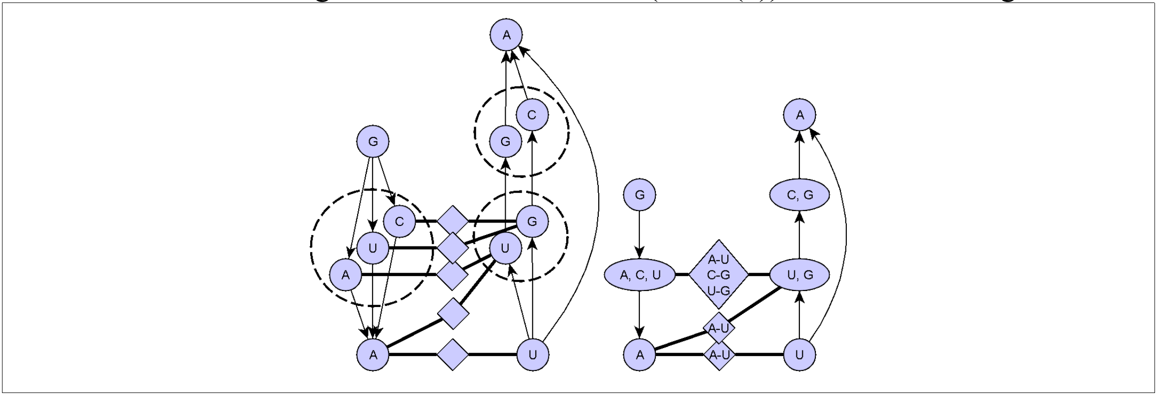


*Table 2: Profile nodes for RNAPOAs. Left: a segment of a string-RNAPOA. Right: a profile string-RNAPOA still contains alternate paths for indels and base pair shifts, but merges mismatched bases.*

The algorithms presented in this thesis charge the same cost for each indel'd base, but this is not biologically accurate. It should be straightforward to use affine indel costs for the RNAPOA algorithms, since they are based on total order alignment algorithms that have affine gap variants.[29] It is also straightforward to modify RNAPOAs to use position-specific substitution and indel costs, which can greatly improve alignment accuracy for sequences with low similarity (Thompson *et al.*, 1994). It remains to be seen whether partial order alignments can support more advanced editing models such as logarithmic gap penalties (Gu and Li, 1995) or different gap penalties within regions of paired bases (Klein and Eddy, 2003). In addition, to rigorously align phylogenetically related RNAs using base pairing probabilities, we need a model of how to combine probabilities based on thermodynamic predictions (as used by e.g. (Hofacker et al, 2004)) with log-odds scoring rules based on phylogenetic comparisons (as used by e.g. (Klein and Eddy, 2003)). Another concern is that scoring models developed for total order alignments have different statistical properties for partial order alignments. As (Lee *et al.*, 2002) note, adding a random sequence to a partial order alignment can only increase the number of alternate paths, and hence can only improve the scores of future alignments. If we want to test the statistical significance of an alignment, e.g. whether a query sequence is better than a random sequence at matching a family of aligned RNAs, we need to develop a different type of scoring function.

However, before attempting to improve the speed or scoring functions of RNAPOAs, they should be fully implemented and tested on more interesting datasets. Not only will this establish give the properties of the RNAPOA algorithms an empirical grounding, but it will help determine which avenues for improvement are most promising. This will also allow an investigation into how different schemes for constructing multiple alignments interact with the properties of RNA structures and partial-order alignments built from various types of datasets (phylogenetically determined structures, thermodynamic predictions, base pairing probabilities, etc.).

RNAPOAs are rather complex graphs, so it is important to display them in a way that makes their information accessible. I used the generic graph editor `yEd` to draw the images in this thesis, but RNAPOAs required considerable manual tweaking to make them presentable. This would quickly become tedious in a research environment, especially for large alignments. There are quite a few programs specialized for drawing RNA structures, but they typically draw only one structure at a time and don't allow tertiary structure. In contrast, RNAPOAs may represent a large ensemble of tertiary structures with many alternate pairs and alternate successors for each base! Of course, we could always translate RNAPOAs into total-order, row-column alignments, but graph drawings are a more intuitive representation for RNA structure. In short, it may actually be more challenging to design interfaces for drawing and editing RNAPOAs than it was to design algorithms to align them.

29 (Wang and Zhang, 2001) for arc-annotated strings, (Wang and Zhao, 2003) for ordered trees.

## 6. References

Akutsu T. (2000) Dynamic programming algorithms for RNA secondary prediction with pseudoknots. Discrete Applied Mathematics 104, 45-62

Altschul S.F., Madden T.L., Schäffer A.A. (1997): Gapped BLAST and PSIBLAST: a new generation of protein database search programs. *Nucleic Acids Res.* **25**,3389-3402.

Bafna,V., Muthukrishnan,S., Ravi,R. (1995) Computing Similarity between RNA Strings. CPM 1995, 1-16.

Bafna, V. and Zhang, S. (2004). FastR: Fast database search tool for non-coding RNA. Proceedings of IEEE Computational Systems Bioinformatics (CSB) Conference, 2004 :52-61.

Bahr A, Thompson JD, Thierry JC, Poch O. (2001) BAliBASE (Benchmark Alignment dataBASE): enhancements for repeats, transmembrane sequences and circular permutations. Nucleic Acids Res. 2001 Jan 1;29(1):323-6.

Bellgard M, Gamble T, Reynolds M, Hunter A, Trifonov E, Taplin R. (2003) Gap mapping: a paradigm for aligning two sequences. Appl Bioinformatics. 2(3 Suppl):S31-5.

Bille, P. (2003) Tree Edit Distance, Alignment Distance and Inclusion. Technical report TR-2003-23 in IT University Technical Report Series, March 2003.

Bonhoeffer, S., McCaskill, J.S., Stadler, P.F., and Schuster, P., (1993) RNA multi-structure landscape: A study based on temperature dependent partition functions, *Eur. Biophys. J.*, 22:14–24, 1993.

Bromberg-Martin, E., Kasprzak, W., Shapiro, B.A. A system for data mining large numbers of RNA secondary structures. currently unpublished.

Brown, J.W. (1999) The Ribonuclease P Database. Nucleic Acids Research 27:314.

Cannone J.J., Subramanian S., Schnare M.N., Collett J.R., D'Souza L.M., Du Y., Feng B., Lin N., Madabusi L.V., Muller K.M., Pande N., Shang Z., Yu N., and Gutell R.R. (2002). The Comparative RNA Web (CRW) Site: An Online Database of Comparative Sequence and Structure Information for Ribosomal, Intron, and other RNAs. BioMed Central Bioinformatics. 3:15.

Dayhoff,M.O., Schwartz, R.M. & Orcutt, B.C. (1978) A model of evolutionary change in proteins. In Atlas of Protein Sequence and Structure, vol 5. Suppl. 3. Pp. 345-352, National Biomedical Research Foundation, Washington, DC.

Deogun, J.S., Yang, J., Ma, F. (2004) EMAGEN: An Efficient Approach to Multiple Whole Genome Alignment. APBC 2004: 113-122

Feng,D. and Doolittle,R. F. (1987) Progressive sequence alignment as a prerequisite to correct phylogenetic trees. J. Mol. Evol. 25:351-360

Fitch, W.M. and Margoliash. (1967) E. Construction of phylogenetic trees, Science, 155, 279--284.

Gardner PP & Giegerich R (2004) A comprehensive comparison of comparative RNA structure prediction approaches. BMC Bioinformatics.5(1):140.

Gotoh, O. (1982). An improved algorithm for matching biological sequences. J. Mol. Biol. 162, 705-708.

Grasso,C., and Lee,C. (2004) Combining Partial Order Alignment and Progressive Multiple Sequence Alignment Increases Alignment Speed and Scalability to Very Large Alignment Problems. Bioinformatics 20:1546-1556.

Grasso,C., Quist,M., Ke,K., and Lee,C. (2003) POAVIZ: A Partial Order Multiple Sequence Alignment Visualizer. Bioinformatics 19: 1446-1448.

Gu X., Li W.-H. (1995). The size distribution of insertions and deletions in human and rodent pseudogenes suggest the logarithmic gap penalty for sequence alignments. J. Mol. Evol. 40:464-473.

Henikoff S, Henikoff JG. (1992) Amino acid substitution matrices from protein blocks. Proc Natl Acad Sci U S A. 1992 Nov 15;89(22):10915-9.

Higgins DG, Sharp PM. (1988) CLUSTAL: a package for performing multiple sequence alignment on a microcomputer. Gene 73:237-44

Hofacker,I., Bernhart,S., and Stadler,P. (2004) Alignment of RNA base pairing probability matrices. Bioinformatics. 2004 Sep 22;20(14):2222-7.

Hofacker,I., Fekete,M., and Stadler,P. (2002) Secondary structure prediction for aligned RNA sequences. Journal of Molecular Biology, 319:1059-66.

Hofacker,I., Fontana,W., Stadler,P., Bonhoeffer,L.S., Tacker,M., and Schuster,P. (1994) Fast Folding and Comparison of RNA Secondary Structures. Monatsh.Chem., 125, 167-188.

Höchsmann,M., Toller,T., Giegerich,R., and Kurtz,S. (2003) Local similarity in RNA secondary structures. Proceedings of CS Bioinformatics, IEEE Computer Society Press, 159-168.

Höchsmann,M., Voss,B., and Giegerich,R. (2004) Pure Multiple RNA Secondary Structure Alignments: A Progressive Profile Approach. IEEE/ACM Transactions on Computational Biology and Bioinformatics, 1, 53-62.

Jansson, J., Ngo, N.T., Sung, W-K. (2004) Local Gapped Subforest Alignment and Its Application in Finding RNA Structural Motifs. ISAAC 2004: 569-580.

Jiang, T., Wang, L., and Zhang, K. (1995) Alignment of trees - an alternative to tree edit. Theoretical Computer Science (TCS), 143.

Jiang,T., Lin,G-H., Ma,B., and Zhang,K. (2002). A General Edit Distance between RNA Structures. Journal of Computational Biology 9(2): 371-388.

Katoh K, Misawa K, Kuma K, Miyata T. (2002). MAFFT: a novel method for rapid multiple sequence alignment based on fast Fourier transform. Nucleic Acids Res. 30(14), pg. 3059-66.

Klein, R. and Eddy, S. (2003) RSEARCH: Finding homologs of single structured RNA sequences. BMC Bioinformatics 2003, 4:44.

Lee,C., Grasso,C., and Sharlow,M. (2002) Multiple Sequence Alignment Using Partial Order Graphs. Bioinformatics 18: 452-464.

Liu, J., Wang, J., Hu, J., and Tian, B. (2005) A method for aligning RNA secondary structures and its application to RNA motif detection. *BMC Bioinformatics* 6:89.

Lyngsø, R.B., Zuker, M., and Pedersen, C.N.S. (1999) An improved algorithm for RNA secondary structure prediction. Tech. report BRICS-RS-99-15, Aarhus Univ., Datalogisk afdeling.

Moulton,V., Zuker,M., Steel,M., Pointon,R., Penny,D. (2000) Metrics on RNA secondary structures. Journal of Computational Biology, 7, 277-292.

Needleman SB, Wunsch CD. (1970) A general method applicable to the search for similarities in the amino acid sequence of two proteins. J Mol Biol. Mar;48(3):443-53.

Notredame C., Higgins D. G., Heringa J. (2000) T-Coffee: A novel method for fast and accurate multiple sequence alignment. J. Mol. Biol. 302:205-17

Raphael, B., Zhi, D., Tang, H. and Pevzner, P. (2004). A novel method for multiple alignment of sequences with repeated and shuffled elements. Genome Research 14, 2336-2346.

Saitou,N., and Nei,M. (1987) The neighbor-joining method: a new method for reconstructing phylogenetic trees. Mol Biol Evol., 4, 406-25.

Sankoff, D. (1985). Simultaneous solution of the RNA folding, alignment, and proto-sequence problems. SIAM J. Appl. Math., 45, 810–825.

Schneider, T.D. and Stephens, R.M. (1990) Sequence Logos: A New Way to Display Consensus Sequences, NAR 18:6097-6100.

Shapiro,B.A. (1988) An algorithm for comparing multiple RNA secondary structures. Comput. Appl. Biosci., 4, 387-393.

Shapiro,B.A. and Zhang,K. (1990) Comparing multiple RNA secondary structures using tree comparisons. Comput. Appl. Biosci., 6, 309-318.

Siebert,S., and Backofen,R. (2003) MARNA: A Server for Multiple Alignment of RNAs. In Proceedings of the German Conference on Bioinformatics, pp. 135-140.

Smith, T.F. and Waterman, M.S. (1981) Identification of common molecular subsequences, *J. Mol. Biol.*, 147(1):195–197.

Thompson, J. D., Higgins, D. G., and Gibson, T. J. (1994) CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. Nucleic Acids Res 22:4673-4680.

Wang, L. and Jiang, T. (1994) On the complexity of multiple sequence alignment. *J Comp. Biol.*, 1 (4), 337-348.

Wang, L., and Zhao, J. (2003) Parametric alignment of ordered trees. Bioinformatics Vol. 19 no. 17, pages 2237-2245

Wang, Z., and Zhang, K. (2001) Alignment between two RNA Structures. MFCS 2001, pags 690-702.

Wang, Z., and Zhang, K. (2004) Multiple RNA Structure Alignment. CSB 2004, pp. 246-254

Yang, Q. and Blanchette, M. (2004) StructMiner: A Tool for Alignment and Detection of Conserved Secondary Structure. *Genome Informatics*, 15(2), 102-111.

Zuker,M., Mathews,D.H., and Turner,D.H. (1999) Algorithms and Thermodynamics for RNA Secondary Structure Prediction: A Practical Guide. In Barciszewski,J. And Clark,B.F.C. (eds), RNA Biochemistry and Biotechnology, NATO ASI Series, Kluwer Academic Publishers.