A Simple, Practical and Complete $O(\frac{n^3}{\log n})$-Time Algorithm for RNA Folding Using the *Four-Russians* Speedup

Yelena Frid and Dan Gusfield

Department of Computer Science, U.C. Davis

Abstract. The problem of computationally predicting the secondary structure (or folding) of RNA molecules was first introduced more than thirty years ago and yet continues to be an area of active research and development. The basic *RNA-folding problem* of finding a maximum cardinality, non-crossing, matching of complimentary nucleotides in an RNA sequence of length n, has an $O(n^3)$-time dynamic programming solution that is widely applied. It is known that an $o(n^3)$ worst-case time solution is possible, but the published and suggested methods are complex and have not been established to be practical. Significant practical improvements to the original dynamic programming method have been introduced, but they retain the $O(n^3)$ worst-case time bound when n is the only problem-parameter used in the bound. Surprisingly, the most widely-used, general technique to achieve a worst-case (and often practical) speed up of dynamic programming, the *Four-Russians* technique, has not been previously applied to the RNA-folding problem. This is perhaps due to technical issues in adapting the technique to RNA-folding.

In this paper, we give a simple, complete, and practical Four-Russians algorithm for the basic RNA-folding problem, achieving a worst-case time-bound of $O(n^3/\log(n))$. We show that this time-bound can also be obtained for richer nucleotide matching scoring-schemes, and that the method achieves consistent speed-ups in practice. The contribution is both theoretical and practical, since the basic RNA-folding problem is often solved multiple times in the inner-loop of more complex algorithms, and for long RNA molecules in the study of RNA virus genomes.

1 Introduction

The problem of computationally predicting the secondary structure (or folding) of RNA molecules was first introduced more than thirty years ago ([9,8,14,11]), and yet continues to be an area of active research and development, particularly due to the recent discovery of a wide variety of new types of RNA molecules and their biological importance. Additional interest in the problem comes from synthetic biology where modified RNA molecules are designed, and from the study of the complete genomes of RNA viruses (which can be up to 11,000 basepairs in length).

The basic *RNA-folding problem* of finding a maximum cardinality, non-crossing, matching of complimentary nucleotides in an RNA sequence of length n, is at the heart of almost all methods to computationally predict RNA secondary structure, including more complex methods that incorporate more realistic folding models, such as allowing some crossovers (pseudoknots). Correspondingly, the basic $O(n^3)$-time *dynamic-programming* solution to the RNA-folding problem remains a central tool in methods to predict RNA structure, and has been widely exposed in books and surveys on RNA folding, computational biology, and computer algorithms. Since the time of the introduction of the $O(n^3)$ dynamic-programming solution to the basic RNA-folding problem, there have been several practical heuristic speedups ([2,12]); and a complex, worst-case speedup of $O(n^3(loglogn)/(logn)^{1/2})$ time ([1]) whose practicality is unlikely and unestablished. In [2], Backofen et al. present a compelling, practical reduction in space and time using the observations of [12] that yields a worst-case improvement when additional problem parameters are included in the time-bound i.e. $O(nZ)$ were $n \leq Z \leq n^2$. The method however retains an $\mathrm{O}(n^3)$ time-bound when only the length parameter n is used[1].

Surprisingly, the most widely-used and known, general technique to achieve a worst-case (and often practical) speed up of dynamic programming, the Four-Russians technique, has not been previously applied to the RNA-folding problem, although the general Four-Russians technique has been cited in some RNA folding papers. Two possible reasons for this are that a widely exposed version of the original dynamic-programming algorithm does not lend itself to application of the Four-Russians technique, and unlike other applications of the Four-Russians technique, in RNA folding, it does not seem possible to separate the preprocessing and the computation phases of the Four-Russians method; rather, those two phases are interleaved in our solution.

In this paper, we give a simple, complete and practical Four-Russians algorithm for the basic RNA-folding problem, achieving a worst-case time reduction from $O(n^3)$ to $O(n^3/log(n))$. We show that this time-bound can also be obtained for richer nucleotide matching scoring-schemes, and that the method achieves significant speed-ups in practice. The contribution is both theoretical and practical, since the basic RNA-folding problem is often solved multiple times in the inner-loop of more complex algorithms and for long RNA molecules in the study of RNA virus genomes.

Some of technical insights we use to make the Four-Russians technique work in the RNA-folding dynamic program come from the paper of Graham et. al. ([5])

[1] Backofen et al. ([2]) also comment that the general approach in ([1]) can be sped up by combining a newer paper on the all-pairs shortest path problem ([3]). That approach, if correct, would achieve a worst-case bound of $(O(\frac{n^3 * \log^3(log(n))}{log^2 n}))$ which is below the $O(n^3/\log n)$ bound established here. But that combined approach is highly complex, uses word tricks, is not fully exposed, and has practicality that is unestablished and not promising (in our view).

which gives a Four-Russians solution to the problem of Context-Free Language recognition[2].

2 A Formal Definition of the Basic RNA-Folding Problem

The input to the basic RNA-folding problem consists of a string K of length n over the four-letter alphabet {A,U,C,G}, and an optional integer d. Each letter in the alphabet represents an RNA *nucleotide.* Nucleotides A and U are called *complimentary* as are the nucleotides C and G. A *matching* consists of a set M of *disjoint* pairs of sites in K. If pair (i,j) is in M, then the nucleotide at site i is said to *match* the nucleotide at site j. It is also common to require a fixed minimum distance, d, between the two sites in any match. A match is a *permitted match* if the nucleotides at sites i and j are complimentary, and $|i-j| > d$.[3] A matching M is *non-crossing* or *nested* if and only if it does not contain any four sites $i < i' < j < j'$ where (i,j) and (i',j') are matches in M. Graphically, if we place the sites in K in order on a circle, and draw a straight line between the sites in each pair in M, then M is non-crossing if and only if no two such straight lines cross. Finally, a *permitted matching* M is a matching that is non-crossing, where each match in M is a permitted match. The basic RNA-folding problem is to find a permitted matching of *maximum cardinality.* In a richer variant of the problem, an n by n *integer scoring matrix* is given in the input to the problem; a match between nucleotides in sites i and j in K is given score $B(i,j)$. The problem then is to find a matching with the largest total score. Often this scoring scheme is simplified to give a constant score for each permitted A,U match, and a different constant score for each permitted C,G match.

3 The Original $O(n^3)$ Time Dynamic Programming Solution

Let $S(i,j)$ represent the score for the optimal solution that is possible for the subproblem consisting of the sites in K between i and j inclusive (where $j > i$).

Then the following recurrences hold: $S(i,j) = \max\{ \underbrace{S(i+1,j-1)+B(i,j)}_{\textbf{rule a}},$

$\underbrace{S(i,j-1)}_{\textbf{rule b}}, \underbrace{S(i+1,j)}_{\textbf{rule c}}, \underbrace{Max_{i<k<j} S(i,k) + S(k+1,j)}_{\textbf{rule d}} \}$

[2] Note that although it is well-known how to reduce the problem of RNA folding to the problem of *stochastic* context-free parsing ([4]), there is no known reduction to non-stochastic context-free parsing, and so it is not possible to achieve the $O(n^3/\log n)$ result by simply reducing RNA folding to context-free parsing and then applying the Four-Russians method from ([5]).

[3] We let d=1 for the remainder of the paper for simplicity of exposition, but in general, d can be any value from 1 to n.

Rule *a* covers all matchings that contain an (i, j) match; Rule *b* covers all matchings when site j is not in any match; Rule *c* covers all matchings when site i is not in any match; Rule *d* covers all matchings that can be decomposed into two non-crossing matchings in the interval $i..k$, and the interval $k + 1..j$.

In the case of Rule *d*, the matching is called a *bipartition*, and the interval $i..k$ is called the **head** of bipartition, and the interval $k + 1..j$ is the called the **tail** of the bipartition.

These recurrences can be evaluated in nine different ordering of the variables i,j,k [13]. A common suggestion [13,7] is to evaluate the recurrences in order of increasing distance between i and j. That is, the solution to the RNA folding problem is found for all substrings of K of length two, followed by all substrings of length three, etc. up to length n. This dynamic programming solution is widely published in textbooks, and it is easy to establish that it is correct and that it runs in $O(n^3)$ worst-case time. However, we have not found it possible to apply the Four-Russians technique using that algorithmic evaluation order, but will instead use a different evaluation order.

4 An Alternative $O(n^3)$-Time Dynamic Programming Solution

```
for $j$ =2 to $n$ do
    [Independent] Calculations below don't depend on the current column j
    for $i$ =1 to $j - 1$ do
        S(i,j)=max( S(i+1,j-1)+B(i,j) , S(i,j-1)) (Rules a, b )
    [Dependent] Calculations below depend on the current column j
    for $i$ =$j - 1$ to 1 do
        S(i,j)=max(S(i+1,j) , S(i,j) ) (Rule c)
        for $k = j - 1$ to i+1 do {The loop below is called the Rule d loop}
            S(i,j)=max( S(i,j), S(i,k-1)+S(k,j) ) (Rule d)
```

The recurrences used in this algorithm are the same as before, but the order of evaluation of S(i,j) is different. It is again easy to see that this Dynamic Programming Algorithm is correct and runs in $O(n^3)$ worst-case time. We will see that this Dynamic Programming algorithm can be used in a Four-Russians speed up.

5 The Four-Russians Speedup

In the Second Dynamic Programming algorithm, each execution of the loop labeled "independent" takes $O(n)$ time, and is inside a loop that executes only $O(n)$ times, so the independent loop takes $O(n^2)$ time in total, and does not need any improvement. The cubic-time behavior of the algorithm comes from the fact that there are three nested loops, for j, i and k respectively, each incrementing $O(n)$ times when entered. The speed-up we will obtain will be due to reducing the work in the Rule d loop. Instead of incrementing k through each value from $j-1$ down to $i+1$, we will combine indices into groups of size q (to be determined later) so that only constant time per group will be needed. With that speed up, each execution of that Rule d loop will increment only $O(n/q)$ times when entered. However, we will also need to do some preprocessing, which takes time that increases with q. We will see that setting $q = \log_3(n)$ will yield an $O(n^3/\log n)$ overall worst-case time bound.

5.1 Speeding Up the Computation of S

We now begin to explain the speed-up idea. For now, assume that $B(i,j) = 1$ if (i,j) is a permitted match, and is 0 otherwise. First, conceptually, divide each *column* in the S matrix into groups of q rows, where q will be determined later. For this part of the exposition, suppose $j-1$ is a multiple of q, and let rows $1...q$ be in a group called Rgroup 0, rows $q+1..2q$ be in Rgroup 1 etc, so that rows $j-q...j-1$ are Rgroup $\lfloor j-1\rfloor/q$. We use g as the index for the groups, so g ranges from 0 to $(\lfloor j-1\rfloor/q)$. See Figure 1. We will modify the Second Dynamic Program so that for each fixed i,j pair, we do not compute Rule d for each $k = j-1$ down to $i+1$. Rather, we will only do constant-time work for each Rgroup g that falls completely into the interval of rows from $j-1$ down to $i+1$. For the (at most) one Rgroup that falls partially in that interval, we execute the Rule d loop as before. Over the entire algorithm, the time for those partial intervals is $O(n^2q)$, and so this detail will be ignored until the discussion of the time analysis.

Introducing vector V_g and modified Rule d loop. We first modify the Second Dynamic Program to accumulate auxiliary vectors V_g inside the Rule d loop. For a fixed j, consider an Rgroup g consisting of rows $z, z-1, z-q+1$, for some $z < j$, and consider the associated consecutive values $S(z,j), S(z-1,j)...S(z-q+1,j)$. Let V_g be the vector of *those* values in that order.

The work to accumulate V_g may seem wasted, but we will see shortly how V_g is used.

Introducing v_g. It is clear that for the simple scoring scheme of $B(i,j) = 1$ when (i,j) is a permitted match, and $B(i,j) = 0$ when (i,j) is not permitted, $S(z-1,j)$ is either equal to S(z,j) or is one more then S(z,j). This observation holds for each consecutive pair of values in V_g. So for a single Rgroup g in column j, the change in consecutive values of V_g can be encoded by a vector of length $q-1$, whose values are either 0 or 1. We call that vector v_g. We therefore define the function *encode*:$V_g \rightarrow v_g$ such that v_g[i]=V_g[i-1]-V_g[i]. Moreover, for any fixed j, immediately after all the S values have been computed for the cells in an Rgroup g, function *encode*(V_g) can be computed and stored in $O(q)$ time, and v_g will then be available in the Rule d loop for all i smaller than the smallest row in g. Note that for any fixed j, the time needed to compute all the encode functions is just $O(n)$.

Introducing Table R and the use of v_g. We examine the action of the Second Dynamic-Programming algorithm in the Rule d loop, for fixed j and fixed $i < j - q$. For an Rgroup g in column j, let $k^*(i, g, j)$ be the index k in Rgroup g such that $S(i, k-1) + S(k, j)$ is maximized, and let $S^*(i, g, j)$ denote the actual value $S(i, k^*(i, g, j) - 1) + S(k^*(i, g, j), j)$.

Note that the Second Dynamic-Program can find $k^*(i, g, j)$ and $S^*(i, g, j)$ during the execution of the Rule d loop, but would take $O(q)$ time to do so. However, by previously doing some preprocessing (to be described below) before column j is reached, we will reduce the work in each Rgroup g to $O(1)$ time.

To explain the idea, suppose that before column j is reached, we have *precomputed* a table R which is indexed by i, g, v_g. Table R will have the property that, for fixed j and $i < j-q$, a *single lookup* of $R(i, g, v_g))$ will effectively return $k^*(i, g, j)$ for any g. Since $k - 1 < j$ and $k > i$, both values $S(i, k^*(i, g, j) - 1)$ and $S(k^*(i, g, j), j)$ are known when we are trying to evaluate $S(i, j)$, so we can find $S(i, k^*(i, g, j) - 1) + S(k^*(i, g, j), j)$ in $O(1)$ operations once $k^*(i, g, v_g)$ is known.

Since there are $O(j/q)$ Rgroups, it follows that for fixed j and i, by calling table R once for each Rgroup, only $O(j/q)$ work is needed in the Rule d loop. Hence, for any fixed j, letting i vary over its complete range, the work will be $O(j^2/q)$, and so the total work (over the entire algorithm) in the Rule d loop will be $O(n^3/q)$. Note that as long as $q < n$, some work has been saved, and the amount of saved work increases with increasing q. This use of the R table in the Rule d loop is summarized as follows:

Dependent section using table R.

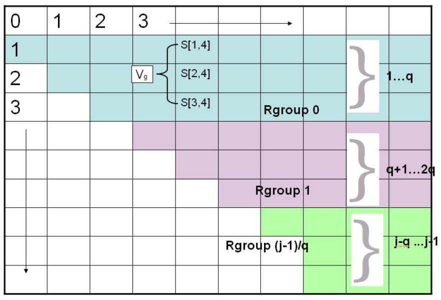
```
for $g = \lfloor (j-1)/q \rfloor$ to $\lfloor (i+1)/q \rfloor$ do
    retrieve $v_g$ given $g$
    retrieve $k^*(i, g, j)$ from $R(i, g, v_g)$
    S(i,j)=max( S(i,j), $S(i, k^*(i, g, j) - 1) + S(k^*(i, g, j), j)$ );
```

Of course, we still need to explain how R is precomputed.

5.2 Obtaining Table *R*

Before explaining exactly where and how table R is computed, consider the action of the Second Dynamic-Programming algorithm in the Rule d loop, for a fixed j. Let g be an Rgroup consisting of rows $z - q + 1, z - q, ..., z$, for some $z < j$. A key observation is that if one knows the single value $S(z, j)$ and the entire vector v_g, then one can determine all the values $S(z - q + 1, j)...S(z, j)$ or V_g. Each such value is exactly $S(z, j)$ plus a partial sum of the values in v_g. In more detail, for any $k \in g$, $S(k, j) = S(z, j) + \sum_{p=0}^{p=z-k-1} v_g[p]$. Let *decode*$(v_g)$ be a function that returns the vector V' where V'[k]=$\sum_{p=0}^{p=z-k-1} v_g[p]$.

Next, observe that if one does *not* know any of the V_g in the rows of g (e.g., the values $S(z - q + 1, j), S(z - 1, j)...S(z, j)$), but *does* know all of v_g, then, for any fixed i below the lowest row in Rgroup g (i.e., row z-q+1), one can find the value of index k in Rgroup g to maximize $S(i, k - 1) + S(k, j)$. That value of k is what we previously defined as $k^*(i, g, j)$. To verify that $k^*(i, g, j)$ can be



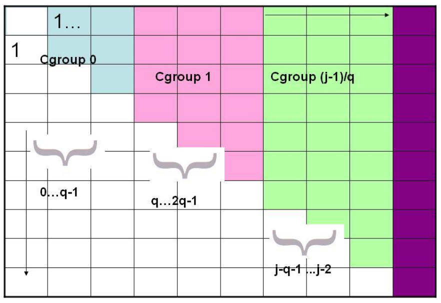


Fig. 1. Rgroups with q=3

Fig. 2. Cgroups

determined from v_g, but without knowing any S values in column j, recall that since $k-1 < j$, $S(i, k-1)$ is already known. We call this Fact 1.

Precomputing the *R* table. We now describe the preprocessing that is needed to compute table R.

Conceptually divide matrix for S into groups of *columns* of size q, i.e., the same size groups that divide each column. Columns 1 through $q-1$ are in a group we call Cgroup 0, q through $2q-1$ are in Cgroup 1 etc, and we again use g to index these groups.

Assume we run the Second Dynamic Program until j reaches $q-1$. That means that all the $S(i,j)$ values have been completely and correctly computed for all columns in Cgroup 0. At that point, we compute the following:

```
for each binary vector $v$ of length $q-1$ do
    $V'$=decode(v)
    for each $i$ such that $i < q-1$ do
        $R(i,0,v)$ is set to the index $k$ in Rgroup 0 such that S(i,k-1) + $V'$[k]
        is maximized. {we let $k^*$ denote that optimal $k$ }
```

The above details the preprocessing done after all the S values in Cgroup 0 have been computed. In general, for Cgroup $g > 0$, we could do a similar preprocessing after all the entries in columns of Cgroup g have been computed. That is, k^*(i,g,v) could be found and stored in R(i,g,v) for all $i < g * q$.

This describes the preprocessing that is done after the computation of the S values in each Rgroup g. With that preprocessing, the table R is available for use when computing S values in any column $j > g \times q$. Note that the preprocessing computations of table R are interleaved with the use of table R. This is different than other applications of the Four-Russians technique that we know of. Note also that the amount of preprocessing work increases with increasing q. Several additional optimizations are possible, of which one, parallel computation, is described in Section 5.5.

With this, the description of the Four-Russians speed up is complete. However, as in most applications of the Four-Russians idea, q must be chosen carefully. If not chosen correctly, it would seem that the time needed for preprocessing would be greater than any saving obtained later in the use of R. We will see in the next section that by choosing q=$log_3(n)$ the overall time bound is $O(n^3/\log(n))$.

5.3 Pseudo-code for RNA Folding Algorithm with Four Russians SpeedUp

```
for $j$ =2 to $n$ do
    [Independent] Calculations below don't depend on the current column j
    for $i$ =1 to $j-1$ do
        S(i,j)=max( S(i+1,j-1)+B(i,j) , S(i,j-1)) (Rules a, b )
    [Dependent] Calculations below depend on the current column j
    for $i$ =$j-1$ to 1 do
        for $g$ =$\lfloor (j-1)/q \rfloor$ to $\lfloor (i+1)/q \rfloor$ do
            if $(i > k)$, $k \in$ Rgroup $g$ then {this statement runs at most
            once, for the smallest $g$}
                find $k^*(i,g,j)$ by directly computing and comparing S(i,k-
                1)+S(k,j) where $k \in$ L={$g * q$ to $i$}[4] $|L| < q$
            else
                retrieve $v_g$ given $g$
                retrieve $k^*(i,g,j)$ from $R(i,g,v_g)$
                S(i,j)=max( S(i,j), $S(i,k^*(i,g,j)-1)+S(k^*(i,g,j),j)$ );
        if $((i-1) \mod q == 0)$, Compute $v_g$ for group g and store it
    [Table] once Cgroup $g = \lfloor j/q \rfloor$ is complete
    for each binary vector $v$ of length $q-1$ do
        $V'$=decode(v)
        for each $i$ 1 to $i < j-1$ do
            $R(i,g,v)$ is set to the index $k$ in Rgroup g such that
            S(i,k-1) + $V'$[k] is maximized.
```

5.4 Correctness and Time Analysis

From Fact 1, it follows that when the algorithm is in the Rule d loop, for some fixed j, and needs to use R to look up $k^*(i,g,j)$ for some g, $k^*(i,g,j) = R(i,g,v_g)$. It follows immediately, that the algorithm does correctly compute all of the S values, and fills in the complete S table. A standard traceback can be done to extract the optimal matching, in $O(n)$ time if pointers were kept when the S table built.

Time Analysis. The time analysis for any column j can be broken down into the sum of the time analyzes for the **[independent]**, **[dependent]**, **[table]** sections.

For any column j the **[independent]** section of the speedup algorithm remains unchanged from the original algorithm and is O(n) time. For each row

[4] Reversing the order of evaluation in the algorithm for i, and k would eliminate this overhead. Lowering the time for this operation from $0(q)$ to $0(1)$. However for simplicity of exposition we leave those details out.

i, the [**dependent**] section of the speedup algorithm is now broken down into n/q calls to the table R. As discussed above, the total time to compute all the encode functions is $O(n)$ per column, and this is true for all decode functions as well. Therefore in any column j, the dependent section takes $0(\frac{n^2}{q})$ time. Also, the processing of the one (possible) partial group in any column j only takes $O(n^2 q)$ time. The [**table**] section sets R(i,g,v) by computing every binary vector v and then computing and storing the value k*(i,g,v). The variable i ranges from 1 to n and there are 2^{q-1} binary vectors. Hence the dependent section for any j takes $O(n * 2^{q-1})$ time.

In summary, the total time for the algorithm is $O(n^2 * q) + 0(\frac{n^3}{q})$ $+0(n^2 * 2^q)$ time.

Theorem 1. (Runtime) *If* $2 < b < n$, *then the algorithm runs in* $O(n^3/\log_b(n))$ *time.*

Proof. Clearly, $O(n^2 \log_b(n) + \frac{n^3}{q}) = O(n^3/\log_b(n))$.
To show that $n^2 \times 2^{\log_b(n)} = O(n^3/\log_b(n))$ for $2 < b < n$, we need to show that

$$2^{\log_b(n)}{=}O(n/\log_b(n)).$$

The relation holds iff

$$n^{\log_b(2)}{=}O(n/\log_b(n)) \text{ iff}$$
$$\log_b(n) \times n^z = O(n)$$

for $z{=}\log_b(2)$. $0 < z < 1$ since $b > 2$.
The above relation holds if

$$\lim_{n\to\infty} \frac{(n^z * \log_b(n))}{n} = 0$$

We simplify the above equation to $\lim_{n\to\infty} \log_b(n)/(n^{1-z})$
We find the limit by taking the derivative of top and bottom (L' Hopital's Rule)

$$\lim_{n\to\infty} \frac{\log_b(n)}{(n^{1-z})} = \lim_{n\to\infty} \frac{(\frac{1}{n\ln(b)})}{(1-z)n^{-z}} = \lim_{n\to\infty} \frac{1}{n\ln(b)*(1-z)n^{-z}} =$$

$$\lim_{n\to\infty} \frac{1}{(1-z)\ln(b)n^{1-z}} = 0 \text{ since } z < 1.$$

5.5 Parallel Computing

By exploiting the parallel nature of the computation for a specific column, one can achieve a time bound of $O(n^2)$ with vector computing. The [Independent] section computes max(S(i+1,j-1)+B(i,j), S(i,j-1)), and all three of the values are available for all i simultaneously. So for each i we could compute the maximum in parallel with an asymptotic run time of O(1) . The [Dependent] section encodes the entire column into vectors of size q. This could be done in parallel, but sequentially it is has 0(n) asymptotic run time.

Table 1. Empirical Results

Size	$O(n^3)$ Algorithm	$O(n^3/\log(n))$ Algorithm	ratio
1000	3.20	1.43	2.23
2000	27.10	7.62	3.55
3000	95.49	26.90	3.55
4000	241.45	55.11	4.38
5000	470.16	97.55	4.82
6000	822.79	157.16	5.24

The encoding is used to reference the R table. For each i there are q entries in the R table. The maximum for each i computed in parallel takes $O(n+q)$=$O(n)$ time to find. The [Tabling] section can also be done in parallel to find k^* for all possible v vectors in O(1) time, entering all 2^{q-1} values into table R simultaneously. The entire algorithm then takes $O(n^2)$ time in parallel.

6 Generalized Scoring Schemes for B(i,j)

The Four Russians Speed-Up could be extended to any B(i,j) for which all possible differences between S(i,j) and S(i+1,j) do not depend on n. Let C denote the size of the set of all possible differences. The condition that C doesn't depend on n allows one to incorporate not just pairing energy information but also energy information that is dependent on the distance between matching pairs and types of loops. In fact the tabling idea currently can be applied to any scoring scheme as long as the marginal score $(S(i-1,j) - S(i,j))$is not $\Omega(n)$. In this case, the algorithm takes $O(n*(n*C^{q-1} + \frac{n^2}{q} + n)) = O(n^2C^{q-1} + \frac{n^3}{q} + n^2)$ time. If we let $q = \log_b(n)$ with $b > C$, the asymptotic time is again $0(n^3/log(n))$. Based on the proof of Theorem 1, the base of the log must be greater then C in order to achieve the speed up. The scoring schemes in ([6,10]) have marginal scores that are not dependent on n, so the speedup method can be applied in those cases.

6.1 Empirical Results

We compare our Four-Russians algorithm to the original $O(n^3)$-time algorithm. The empirical results shown below give the average time for 50 tests of randomly generated RNA sequences and 10 downloaded sequences from genBank, for each size between 1,000bp and 6,000bp. All times had a standard deviation of .01. The purpose of these empirical results is to show that despite the additional overhead required for the Four-Russians approach, it does provide a consistent practical speed-up over the $O(n^3)$-time method, and does not just yield a theoretical result. In order to make this point, we keep all conditions for the comparisons the same, we emphasize the ratio of the running times rather than the absolute times, and we do not incorporate any additional heuristic ideas or optimizations that might be appropriate for one method but not the other. However, we are aware that there are speedup ideas for RNA folding, which do not reduce the

$O(n^3)$ bound, but provide significant practical speedups. Our empirical results are not intended to compete with those results, or to provide a finished competitive *program*, but to illustrate the practicality of the Four-Russians method, in addition to its theoretical consequences. In future work, we will incorporate all known heuristic speedups, along with the Four-Russians approach, in order to obtain an RNA folding *program* which can be directly compared to all existing methods.

Acknowledgments. This research was partially supported by NSF grants SEI-BIO 0513910, CCF-0515378, and IIS-0803564.

References

1. Akutsu, T.: Approximation and exact algorithms for RNA secondary structure prediction and recognition of stochastic context-free languages. J. Comb. Optim. 3(2-3), 321–336 (1999)
2. Backofen, R., Tsur, D., Zakov, S., Ziv-Ukelson, M.: Sparse RNA folding: Time and space efficient algorithms. In: CPM 2009 (2009)
3. Chan, T.M.: More algorithms for all-pairs shortest paths in weighted graphs. In: STOC, pp. 590–598 (2007)
4. Durbin, R., Eddy, S.R., Krogh, A., Mitchison, G.: Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids. Cambridge University Press, Cambridge (1998)
5. Graham, S.L., Harrison, M., Ruzzo, W.L.: An improved context-free recognizer. ACM Trans. Program. Lang. Syst. 2(3), 415–462 (1980)
6. Hofacker, I.L., Fekete, M., Stadler, P.F.: Secondary structure prediction for aligned RNA sequences. Journal of Molecular Biology 319(5), 1059–1066 (2002)
7. Kleinberg, J., Tardos, E.: Algorithm Design. Addison-Wesley Longman Publishing Co., Inc., Boston (2005)
8. Nussinov, R., Jacobson, A.B.: Fast algorithm for predicting the secondary structure of single-stranded RNA. PNAS 77(11), 6309–6313 (1980)
9. Nussinov, R., Pieczenik, G., Griggs, J.R., Kleitman, D.J.: Algorithms for loop matchings. SIAM Journal on Applied Mathematics 35(1), 68–82 (1978)
10. Seemann, S.E., Gorodkin, J., Backofen, R.: Unifying evolutionary and thermodynamic information for RNA folding of multiple alignments. NAR (2008)
11. Waterman, M.S., Smith, T.F.: RNA secondary structure: A complete mathematical analysis. Math. Biosc. 42, 257–266 (1978)
12. Wexler, Y., Zilberstein, C.B.-Z., Ziv-Ukelson, M.: A study of accessible motifs and RNA folding complexity. Journal of Computational Biology 14(6), 856–872 (2007)
13. Zuker, M., Sankoff, D.: RNA secondary structures and their prediction. Bulletin of Mathematical Biology 46(4), 591–621 (1984)
14. Zuker, M., Stiegler, P.: Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. Nucleic Acids Research 9(1), 133–148 (1981)