Assessing the performance of the dynamic programming approach to RNA folding

Christopher Patton

1. Introduction

Our understanding of the role of RNA in cells has become increasingly complex as researchers continue to make new discoveries. Naturally, eliciting the structure of these molecules is critical to these discoveries. Compared to protein, the biophysical and biochemical laws that govern the secondary and tertiary structure RNA are relatively loose (Kleinberg 2006). In predicting how proteins will fold, there are few possibilities to consider; the shape is well-defined by the chemistry of the component amino acids. The range of possibilities in the structure of RNA necessitates an algorithmic approach to the problem. As a result, RNA folding has been added to the field of computational biology.

There exists a well-known, simple dynamic programming solution for predicting the secondary structure of RNA in order $O(n^3)$ time. The optimal substructure of the problem is based on simple assumptions about the chemistry of nucleotides that make up our model; these assumptions also allow for the polynomial time bound of the algorithm. However, it is reasonable to question how accurately our model predicts secondary structure. In order to move forward with this algorithm, we must qualify the validity of its assumptions.

In the current work, I compare the known secondary structure of a few common tRNA sequences of about the same length to structures predicted by the simple dynamic programming algorithm. I give a high-level description of the dynamic programming approach and provide the corresponding implementation in the C programming language. The results of these tests reveal some flaws in the experimental design, creating uncertainty that we must control for; however, they demonstrate that this simple approach, while a good starting point, is not adequate to predict the shape of RNA molecules. Further, the results suggest that the underlying model of the biological systems must be improved.

2. Background and methods

We begin with an informal view of the problem. Our goal is to abstract the underlying biological forces and develop a model with which we can approach the problem of RNA folding algorithmically. RNA molecules are comprised of a strand of nucleotides: (A)denine, (U)racil, (G)uanine, and (C)ytosine. The basic secondary structure of tRNA, depicted in *Figure 1*, resembles a non-rooted tree with with branches made of base pairs and leaves made of loops with the exception of one loose end, referred to as the accepter stem.

The structure is essentially a sequence of nucleotides folded onto itself; the manner in which the strand folds is dictated by the chemistry of these individual nucleotides and, importantly, the order in which the appear. For example, consider the D-loop in Figure 1, whose sequence is CGAGACAGGACCGACUCG. It is only possible to pair G with C and A with U (Kleinberg 2006). If we change one of the nucleotides in the stem, say CGAGACAGGACCGAC**G**CG, it would not be possible for the D-loop to form , since A does not pair with G. Therefore, the sequence of nucleotides, and the rules that govern the formation of base pairs, dictates the secondary structure.

The biological influences that define the shape of RNA also restrict the size of loops (Kleinberg 2006). Say we modify the sequence so: CGAG**U**CAGGACCGACUCG. If this is the case, U could pair with the A on the other side of the loop, thus shrinking the loop from ten nucleotides to eight. In fact RNA molecules tend to zip together. Because U and A have formed a base pair, the now-adjacent G and C will pair as well. We'll end up with the following loop:

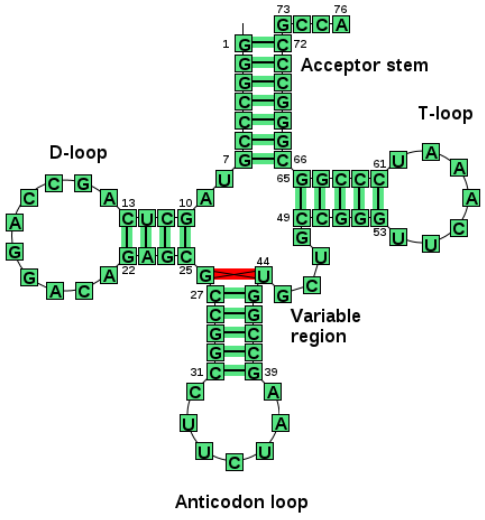

Figure 1: Secondary structure of Aeropyrum permix (from tRNAdb)

```
 CCGA            CC
A    CUCG   →   A  GACUCG
G    GAGC       G  CUGAGC
 GACA            GA
```

Nature imposes a threshold on this process, however; loops may be no smaller than a certain number of nucleotides. (Notice that the acceptor stem is an exception, which contains a non-looping loose strand of RNA.) We call this restriction the *no-sharp-turns* condition (Kleinberg 2006).

Lastly, we assume for the purposes of optimal secondary structure that the strand contains no knots; for example, we assume that for a structure such as `AA … GG … UU … CC`, it is not possible to match `AA` to `UU` and `GG` to `CC`. This *no-crossing* condition is important for our purposes because it provides the foundation of a dynamic programming algorithm, called *optimal substructure;* the optimal structure of a subsequence of a strand of RNA does not depend on the structure of the remainder of the sequence. With respect to the example, the structure of the D-loop is independent of the rest of molecule; in particular, it is not possible for subsequence of the loop to bind with any other part of the molecule, since this would break the *no-crossing* rule. We call this feature of the problem optimal substructure; While there are rare exceptions to this rule in nature, this assumption provides a very good approximation of reality (Kleinberg 2006).

2.1 Assumptions and formal specification

As suggested, these biological consequences lead to a dynamic programming solution. As our objective function, we define the optimal structure of a sequence of RNA as that which forms the highest number of base pairs. We formalize the model developed in the previous section into the following four conditions (Kleinberg 2006):

> Let $X = X_1, X_2, X_3 \ldots X_n$ be a string over the alphabet {`A`, `U`, `G`, `C`}. We denote the secondary structure as a set of tuples $S = \{ (X_i, X_j) \mid i < j \}$, where (X_i, X_j) are subject to the

following restrictions:

(i) (*no-sharp-turns*) Pairs are separated by at least four nucleotides: that is, (X_i, X_j) in the set S implies that $j - i > 4$.

(ii) Allowable pairs are (`A`, `U`), (`U`, `A`), (`G`, `C`), and (`C`, `G`).

(iii) S is a matching: no base appears in more than one pair.

(iv) (*no-crossing*) if (X_i, X_j) in the set S and (X_k, X_l) in the set S, then it is not true that $i < k < j < l$.

(iv) is sufficient to ensure optimal substructure of the solution. Let *OPT(i,j)* denote the objective function, or the maximum number of base pairs over the substring $X_i, \ldots X_j$. If $i < j < k < l$, then the solution space for *OPT(i,j)* and *OPT(k,l)* are distinct. Since we have optimal substructure, we can develop a recurrence for computing the objective function for all subsequences of X.

2.2 A dynamic programming algorithm

Based on these conditions, we can define the following recurrence relation for *OPT*(i,j):

OPT(i,j) = 0 if $j - i < 5$.
OPT(i,j) = max { *OPT(i, j-1)*, 1 + *OPT(i, t-1)* + *OPT(t+1, j-1)* }
for all valid base pairs (X_t, X_j) (Kleinberg 2006).

The intuition for this algorithm is simple. For an interval $X_i, \ldots X_j$, we pick an X_t such that (X_t, X_j) is a valid base pair:

$X_1, X_2, \ldots (X_i, \ldots X_t, \ldots X_j) \ldots X_n$

The score for this choice of *t* is then *OPT(i, t-1)* plus *OPT(t+1, j-1)* plus 1, since (X_t, X_j) is a base pair. To calculate *OPT(i,j)*, we pick the *t* that maximizes this score over this interval. *OPT(i,j)* then gives the highest score for that interval, which we can use for determining the best score for a larger problem, hence the recursion. *OPT(i,j)* = 0 for $j - i > 4$ suffices as our basis.

2.2.1 Implementation[1]

We can produce the optimal structure of a strand of RNA $X = X_1, X_2, \ldots X_n$ by calculating *OPT(1,n)* with this recurrence. We utilize dynamic programming to compute an *n* by n matrix OPT, in which we compute *OPT(i,j)* for all subsequences of X. In order to recover the structure, we'll memeoize our choices of (X_t, X_j) parings along the way. The following psuedocode is an adaption of the algorithm described in Section 6.5 of Kleinberg thatenables us to recover the structure.

> Let each cell of OPT be a 3-tuple *(s, t, j)*, where *s* is the score and *(t, j)* is the base pair chosen for that interval.
>
> **func** *predictRNA* (OPT, X)
> Let $OPT_{i,j} = (0, nil, nil)$ for $j - i < 5$.
> Let $n = |X|$.
> **for** $k \leftarrow 5$ to $n - 1$
> **for** $i \leftarrow 1$ to $n - k$
> $j \leftarrow i + k$
> Max = $OPT_{i,j-1}$
> **for** $t < i$ to $j - 5$
> **if** *(t,j)* is a valid match (2.1) and the score is better than the Max
> then
> Max = (score of *(t, j), t, j*)
> end **if**
> end **for** (*t*)
> $OPT_{i,j}$ = Max
> end **for** (*j*)
> end **for** (*k*)
> return $OPT_{1,n}$
> end **func**

For all *k*-length intervals over the string X of length *n*, compute the optimal score. The optimal score is a maximum value computed by the for-loop over *t*. Recovering the optimal substructure is as follows:

1 An implementation written in C comprises the figures section and is available digitally from the author.

```
func printRNA (OPT, m, n)
  (s, t, j) = OPT_{m,n}
  if s > 0
  then
    print (t, j)
    printRNA (OPT, m, t - 1);
    printRNA (OPT, t + 1, j - 1);
  end if
end func
```

printRNA outputs the indices of the base pairs. As expected, this resembles the recurrence relation in (2.2).

2.2.2 Complexity

predictRNA has three nested for-loops, all roughly bounded by n, the length the string. The algorithm is then clearly order $O(n^3)$.

2.3 Experiments

I ran the algorithm *predictRNA* in (2.2.1) for a number tRNA sequences. These sequences and their known structures were found in the tRNAdb database available online from the University of Leipzig, Germany at http://trna.bioinf.uni-leipzig.de/. The sequences are associated with the organism in which they were discovered. I list the sequence of the molecule, the known structure, and the structure I predicted. Structures are notated as strings, where left and right parenthesis represent the left and right nucleotides of a base pair and dots represent loops or free strands. Note that the sequences in this database contain characters that are not nucleotides: for example “, 1, 2, … 9, L, and P . For the purposes of the current work, alphabetic characters (except A, U, G, and C) were treated as wildcards in forming base pairs; non-alphabetic characters were not aloud to form base pairs.

3. Results

The algorithm's performance varied marginally across all the tests. Most of the time, the program identified key structures of the molecules, such as loops and stems, but the degree of precision

was minimal across the board. I have chosen three of the trials that are representative of the sort of results the algorithm produced.

3.1 Aeropyrum permix

```
GGGCCCGUAGCUCAGCCAGGACAGAGCGCCGGCCUUCUAAGCCGGUGCUGCCGGGUUCAAAUCCCGGCGGGCCCGCCA
(((((((..((((..........)))).(((((.......))))).....(((((.......)))))))))))....  true structure
..((.....))(.....).......(((((((.......))))))....((((((.......))))))).....))..  predicted structure
```

Figure 2: Sequence and structure of Aeropyrum permix (from tRNAdb), depicted in Figure 1

This molecule represents the best results the algorithm produced, in that it predicted the secondary structure of two thirds of the molecule *nearly* perfectly. It identified the shape of the T and Anticodon loops, but failed to elicit the shape of the D-loop. This is likely a result of the fact that it uses nucleotides of the accepter stem in base pairs. This is apparent by the right most end of the strand in the figure.

3.2 Bombyx mori

```
GGGGGCGUALCUCAGADGGUAGAGCRCUCGCJUIGCOP#PGAGAG7UA?CGGGAPCG"UACCCGGCGCCUCCACCA
(((((((..((((........)))).(((((.......))))).....(((((.......)))))))))))....  true structure
(((((((.....).(((((.....))))).....)))).(......).(((((.......))))).....))....  predicted structure
```

Figure 3: Sequence and structure of Bombyx mori (from tRNAdb)

In this case, the algorithm predicted the shape of only one loop correctly. However, the accepter stem and the very beginning of the sequence were predicted correctly. It is interesting as well that the the first seven nucleotides in the sequence were paired properly, but the adjacent structure was completely different. This affect may be due in part to the assumptions made in the implementation. It is possible that the L in the 9th position of the sequence should have been treated as a special case, instead of as a wildcard. This same could perhaps be considered for the '#' in the sequence.

3.2 Ascaris suum

```
GGACGUU""AUAGAUAAGCUAPGCCPAGPUACGKQCPGGGAAGAGAGUCGPCUUCCA
((((((.(..((((.....)))).(((((.......)))))......).))))))....  true structure
..(.......)((.....))((((((.....))))))..((((((......))))))..  predicted structure
```

Figure 4: Sequence and structure of Ascaris suum (from tRNAdb)

This sequence is demonstrative of the worst case results in the experiment. It predicted one loop correctly, but failed for most of the sequence. Here we see a case where there are a large number of “wildcards” in the strand, which I allowed to pair arbitrarily.

4. Discussion

In the implementation of the dynamic programming algorithm for predicting RNA folding, we made our assumptions and developed our matching scheme only considering the nucleotides that make up RNA: A, U, G, and C; other symbols were treated in a generic way. Obviously, these symbols are very meaningful to the shape of the RNA and represent key biological concepts. An improvement to the algorithm must consider these features.

Nevertheless, as we saw in the first two tests, where these affects are less prevalent, the algorithm seemed to predict the shape of the RNA fairly well. However, the results suggests there is definite room for improvement of the assumptions of the algorithm. First, in the implementation, I assumed a minimal loop length of five. In the database, I have observed instances where loops may be as small as four nucleotides long. Oddly enough, letting the minimum loop be equal to five seemed to produce better results upon visual inspection. Second, there is perhaps room to for debate as to whether the chosen objective function (highest number of base pairs) is representative of natural processes. There are perhaps other objectives we can minimize on; for example, we can consider minimizing the amount of free energy in the molecule – instead of maximizing the number of base pairs – without loss of generality.

Pursuing these improvements as well as accounting for the symbols in the database that represent vital biological and chemical features are important steps in furthering this work; however, it would also be a good idea to improve the experimental design. Since the fundamental algorithm deals with strings of nucleotides, we should perhaps consider sequences only containing nucleotides and leave out other symbols. To this end, we can run the algorithm on the DNA genes that could for the RNA sequences. This will give us a good baseline for how the algorithm holds up to the actual structure of molecules and may shed light on the assumptions that need to be improved or changed. We may find as a result that the other non-nucleotide symbols are critical for predicting the structure.

5. Figures

Following is the implementation of *predictRNA* in C. I have also implemented *printRNA* and another function which produces a string representation of the secondary structure. The program is pure C and requires no special libraries. The input sequence is read on standard input. The program should then be used this way:

```
$ cat > sequence.txt
AGCCUUGGCCUCAC
$ gcc -o rna rna.c
$ ./rna < sequence.txt
```

```
/* rna.c
 * RNA secondary structure
 * dynamic programming algorithm
 * 222a - Martel/Gusfield
 *
 * Chris Patton
 * ~13 Oct 2012
 *
 * Dynamic programming algorithm for predicting the secondary structure of
 * RNA. Compute the OPT matrix. The recurrence is as follows:
 *  OPT(i,j) = max {
 *    OPT(i,j-1),
 *    max { 1 + OPT(i,t-1) + OPT(t+1,j-1) }
 *  }
 * for i < j and valid pairing (t,j). The algorithm takes cubic time. Return
 * the number of basepairs in the optimal solution. This approach is from
 * Kleinberg and Tardos, Section 6.5.
 */
```

```
#include "stdio.h"
#include "string.h"
#include "stdlib.h"

#define LOOP 5
#define MAX  256
#define NUCLEOTIDE(x) (x == 'A' || x == 'U' || x == 'G' || x == 'C')
#define ALPHABETIC(x) (('A' <= x && x <= 'Z') || x == '?')

/* cell_t - a cell of the OPT matrix */
typedef struct {
  int score;
  int a, b;
} cell_t;

const cell_t EMPTY_CELL_T = {0, -1, -1};

/* rna_t - the OPT matrix */
typedef struct {
  cell_t opt [MAX][MAX];
  int n;
} rna_t;

void init_rna_t ( rna_t *mol )
/* Initialize a struct rna_t */
{
  int i,j;
  for (i = 0; i < MAX; i++)
    for (j = 0; j < MAX; j++)
      mol->opt[i][j] = EMPTY_CELL_T;
  mol->n = 0;
}


int match( const char *rna, const rna_t *mol, int t, int j )
/* Determine whether (t,j) is a valid base pair. */
{
  char a = rna[t], b = rna[j];
  const cell_t *right = &mol->opt[t+1][j-1];

  /* no sharp-turn condition */
  int r = right->score == 0 ||
    (right->a - t == 1 && j - right->b == 1) ||
    (right->a - t > LOOP && j - right->b > LOOP);

  /* pair condition */
  if (NUCLEOTIDE(a) && NUCLEOTIDE(b))
    return r && (
     (a == 'A' && b == 'U') ||
     (a == 'U' && b == 'A') ||
     (a == 'G' && b == 'C') ||
     (a == 'C' && b == 'G'));

  /* If not nucleotide, but alphabetic, match. Otherwise, don't match. */
  else
    return r && ALPHABETIC(a) && ALPHABETIC(b);
```

```
}

int predict( const char *rna, rna_t *mol )
/* Compute the OPT matrix.
 * Treat alphabetic characters that are not A, U, G, or C as wildcards,
 * but not non-alphabetic characters. */
{
  int i, j, k, t;
  cell_t max, *left, *right;

  init_rna_t (mol);
  mol->n = strlen(rna);
  for (k = LOOP; k < mol->n; k++)
  {
    for (i = 0; i <= mol->n - k; i++)
    /* for all k-length intervals (0, k), (1, k+1), ... (n-k, n) */
    {
      j = i + k;
      max = mol->opt[i][j-1];

      for (t =  i; j - t > LOOP; t++)
      /* for all valid base pairs (t,j), pick the base pair that maximizes
       * the scores for the subproblems OPT(i,t-1) and OPT(t+1,j-1) */
      {
        left  = &mol->opt[i][t-1];
        right = &mol->opt[t+1][j-1];

        if (match (rna, mol, t, j) && left->score + right->score + 1 >
max.score)
        {
          max.score = left->score + right->score + 1;
          max.a = t;    // save the pair so we can recover the structure
          max.b = j;    // later.
        }
      }

      mol->opt[i][j] = max;

    }
  }

  return mol->opt[0][mol->n-1].score;
}

void print_opt (const rna_t *mol)
/* Print the computed OPT matrix */
{
  int i,j;
  for (j = LOOP; j < mol->n; j++) printf("%3d", j);
  for (i = 0; i < mol->n - LOOP; i++)
  {
    printf("\n%-3d", i);
    for (j = LOOP; j < mol->n; j++)
    {
      printf("%3d", mol->opt[i][j].score);
    }
```

```
  }
  printf("\n");
}

void print_pairs (const rna_t *mol, int i, int j)
/* Recover the the optimal secondary structure from the computed OPT matrix.
 * Print the base pairs of the molecule. */
{
  int a = mol->opt[i][j].a,
      b = mol->opt[i][j].b;

  if (a >= 0 && j >= 0)
  {
    printf("(%d, %d)\n", a,b);
    print_pairs (mol, i, a-1);
    print_pairs (mol, a+1, b-1);
  }
}

void make_structure (const rna_t *mol, char *structure, int i, int j)
/* Called by print_structure. Reconstruct the optimal structure
 * and create a friendly string. */
{
  int a = mol->opt[i][j].a,
      b = mol->opt[i][j].b;

  if (a >= 0 && j >= 0)
  {
    structure[a] = '(';
    structure[b] = ')';
    make_structure (mol, structure, i, a-1);
    make_structure (mol, structure, a+1, b-1);
  }

}

void print_structure (const rna_t *mol)
/* Print a string representation of the structure. */
{
  char structure [MAX];
  memset(structure, '.', sizeof(char) * MAX);
  make_structure (mol, structure, 0, mol->n-1);
  if (mol->n == MAX)
    structure[mol->n-1] = '\0';
  else
    structure[mol->n] = '\0';
  printf("%s", structure);
}


int main(int argc, const char **argv)
{

  char c, rna [MAX];
  int i, j=0, score;
  rna_t *molecule = (rna_t *)malloc(sizeof(rna_t));
```

```
  /* Get RNA strand from stdin. Finish input on EOF */
  for (i = 0; (c=getchar()) != EOF && i < MAX; i++)
  {
    switch (c)
    {
      case ' ':
      case '\t':
      case '\n': break;
      case 'A':
      case 'G':
      case 'C':
      case 'U':
      default: rna[j++] = c;
    }
  }
  rna[j] = '\0';

  printf("%s\n", rna);

  /* Compute OPT */
  score = predict( rna, molecule );

  //printf("score: %d\n   ", score);
  //print_opt (molecule);
  print_structure (molecule);
  printf("  score: %d\n", score);
  print_pairs (molecule, 0, strlen(rna)-1);

  free (molecule);
  return 0;
}
```

References

Kleinberg J. and Tardos, E 2006. Algorithm Design: Addison Wesley.

“tRNAdb: Transfer RNA database.” http://trna.bioinf.uni-leipzig.de/DataOutput/, Accessed Oct 2012.