

Supplementary material.

Figures

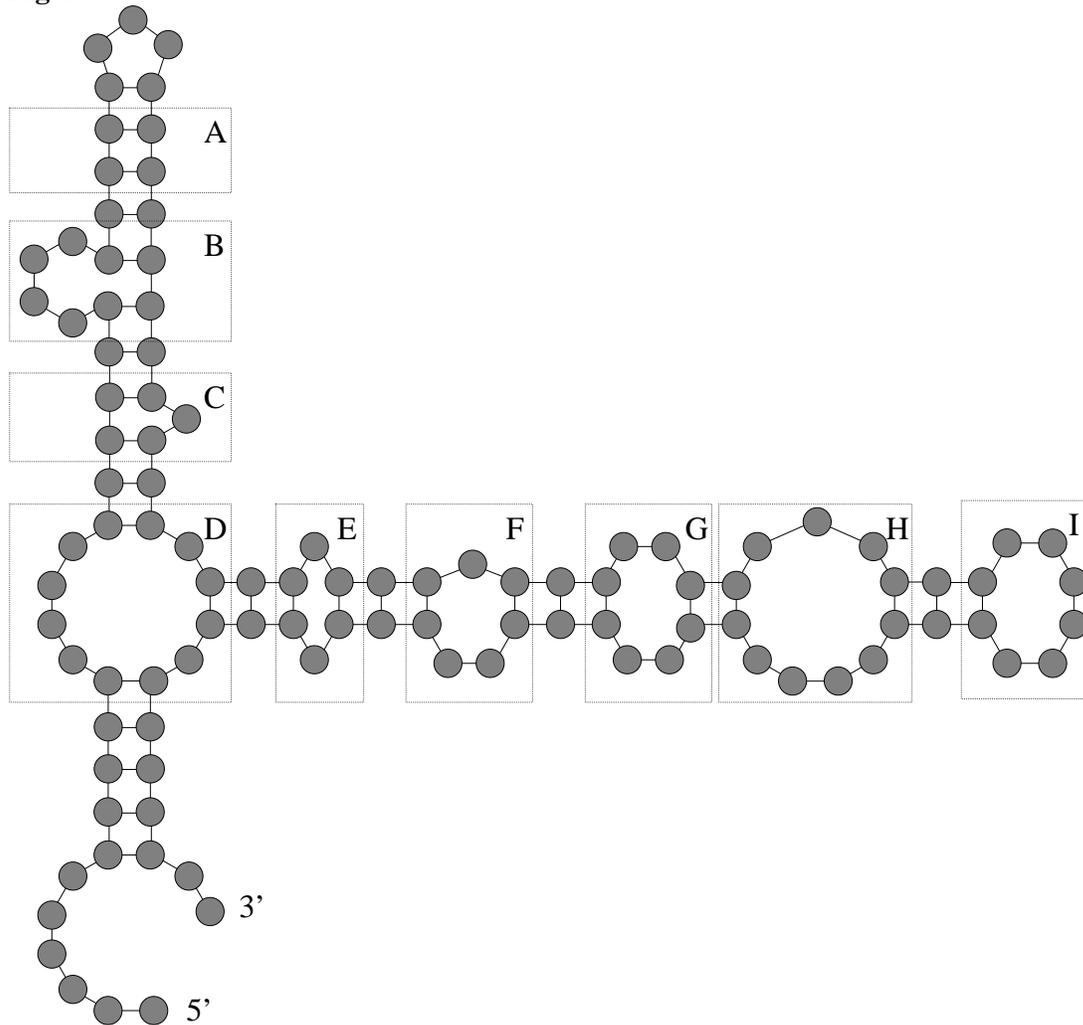


Fig. S1. Example of a secondary structure containing different types of loops: A - stacked loop, B - bulge, C – simple bulge (0x1), D – multi-branched loop, E – simple 1x1 internal loop, F – simple 1x2 internal loop, G – simple 2x2 internal loop, H – internal loop, I – hairpin loop.

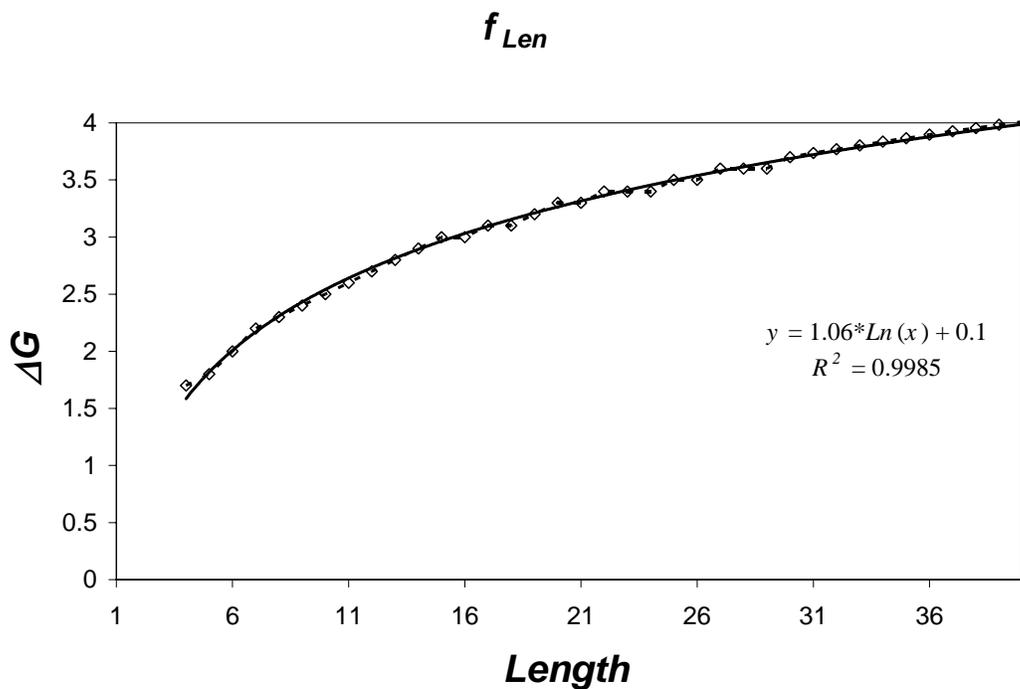


Fig. S2. The graph of function $f_{Len}(t)$ according to Mathews *et al.* (1999) (dotted line) and its logarithmic approximation $y = 1.06 * \ln(x) + 0.1$ (solid line). In Mathews *et al.* (1999) the values of $f_{Len}(t)$ are defined as $c_1 + c_2 * \log(t/30)$ for $t > 30$.

<i>RNA sequence</i>	<i>RNA Length</i>	<i>Max length of a candidate list</i>	<i>Average length of a candidate list</i>
NM_207436	1597	26	2.05
NM_173589	3222	20	1.97
NM_003622	6076	20	2.03
NM_032969	9146	14	2.11
NM_014611	17400	14	1.99

Fig. S4. Lengths of candidate lists within M-algorithm (see sub-section 2.2) for different lengths of RNA.

Algorithms

```
algorithm OptimalRNA(input: RNA[1..L])  
begin  
    // 1. Pre-processing  
    Initialize the data structures  
    // 2. The main loop  
    for all  $B$  from 1 to  $L$  do begin  
2.1.   HairpinRow( $B$ );  
2.2.   SimpleLoopRow( $B$ );  
2.3.   InternalLoopRow( $B$ );  
2.4.   Multi-branchLoopRow( $B$ );  
2.5.   OptimalLoopRow( $B$ );  
    end  
    // 3. Post-processing  
    Restore the optimal RNA secondary structure from the data obtained  
End
```

Algorithm S1. Framework of the algorithm finding the optimal secondary structure of an RNA sequence. At each of the steps 2.1 – 2.5, the function processes the row ROW_B . For each nucleotide pair $(A, B) \in ROW_B$ the function finds the optimal structure with the most external (closing) pair (A, B) , where (A, B) closes the loop of the corresponding type. For example, *Multi-branchLoopRow*(B) corresponds to the structures where the external loop is a multi-branch loop and *SimpleLoopRow*(B) corresponds to the structures having external simple loop, e. g. a stacking pair, a bulge, or an internal loops with small distances between its opening and closing base pairs. The subject of our main interest is function *InternalLoopRow*(B), which finds the optimal internal loops of general form, i.e. having fragments of unpaired bases of length 3 or more. Function *Multi-branchLoopRow*(B) finds the optimal structure for each fragment $[A, B]$ (not necessarily containing the base pairing (A, B)), in addition to finding the optimal structures having external multi-branch loop. Function *OptimalLoopRow*(B) finds, for each pair $(A, B) \in ROW_B$, the best structure with the closing pair (A, B) as the structure having the minimal energy from the four structures obtained in lines 2.1 – 2.4.

```

algorithm InternalLoopRow(input:  $B$ ){
begin
  for all  $B$  from 1 to  $L$  do begin
    InternalLoopMainRow( $B$ ); // Computes  $\Delta G_{Main}(A, B)$  for all pairs  $(A, B) \in ROW_B$ 
    InternalLoopStripRow( $B$ ); // Computes  $\Delta G_{Strip}(A, B)$  for all pairs  $(A, B) \in ROW_B$ 
    InternalLoopFinalRow( $B$ ); // For all pairs  $(A, B) \in ROW_B$  computes
      //  $\Delta G_{Struct}(A, B) =$ 
      //  $= \min\{ base\_value + \Delta G_{Main}(A, B), \Delta G_{Strip}(A, B) \}$ 
  end
end
Algorithm S2. Function InternalLoopRow( $B$ ).

```

```

(a)
algorithm OptimalRNA_G(input: RNA[1..L])
    array VAR_CANDIDATESTRIP[2*L] of list of candidate;
begin
    // 1. Pre-processing
    for all r from 3 to 2*L-1 do begin
        set VAR_CANDIDATESTRIP[r] to empty list;
    end
    // 2. The main loop
    for all B from 1 to L do begin
2.1.   HairpinRow(B);
2.2.   SimpleLoopRow(B);
2.3.   InternalLoopRow_G(B);
2.4.   MultipleLoopRow(B);
2.5.   OptimalLoopRow(B);
2.6.   UpdateCandidate(B);
    end
    // 3. Post-processing
    Restore the optimal RNA secondary structure from the data obtained
end

```

```

(b)
algorithm InternalLoopStripRow_G(input: B){
// Computes  $\Delta G_{Strip}(A, B)$  for all  $(A, B) \in ROW_B$ ;
global var
    float VAR_STRIPWORK[U];
    array VAR_CANDIDATESTRIP[2*L] of list of DAT_CANDIDATE;
local var
    DAT_CANDIDATE C;
begin
    for all base pairs  $(A, B)$  from  $ROW_B$  in descending order by A do begin
         $r = A+B$ ;
         $C = GetFirst(r)$ ;
         $VAR_STRIPWORK[A, B] = C.dG + f_{Len}(B-A) - (C.y - C.x + 2)$ ;
    end
end

```

Algorithm S3.(a) Framework of the algorithm finding the optimal secondary structure of an RNA molecule, modified to implement the function *InternalLoopRowStrip_G*. The call *UpdateCandidate*(*B*). line 2.6, updates lists *VAR_CANDIDATESTRIP*[*r*] from $Cand_{r,B}$ to $Cand_{r,B+1}$ according to newly calculated values $\Delta G(A, B)$ for all $(A, B) \in ROW_B$. The other functions (lines 2.1 – 2.5) give the same results as ones on Supplementary Figure S2.

(b) Function *InternalLoopStripRow_G*. The function *GetFirst*(*r*) gets the 1st element of the list $VAR_CANDIDATESTRIP[r] = CAND_{r,B}$. The data structure *DAT_CANDIDATE* describes one element of the candidate list *x*, *y* and *dG*, are the components of *DAT_CANDIDATE* (see “Description of *G*-algorithm”).

```

algorithm UpdateCandidate(input:  $B$ ){
// Updates value  $VAR\_CANDIDATESTRIP[r]$  from  $Cand_{r,B}$  to  $Cand_{r,B+1}$ .
global var
    array  $VAR\_CANDIDATESTRIP[2*L]$  of list of  $DAT\_CANDIDATE$ ;
local var
    array  $POINTER[2*L]$  of pointers to  $DAT\_CANDIDATE$ ;
     $DAT\_CANDIDATE$   $C$ ;
begin
    1. // Work out sets  $\{(r-B-t, B) \mid t = 0, \dots, width-1\}$ ,
        // see (2.1), Supplementary section 3
    1.1. // Pre-processing of the 1st elements of old candidate lists
        for all base pairs  $(A, B)$  from  $ROW_B$  in ascending order by  $A$  do begin
            for all  $t$  from 0 to  $width-1$  do begin
                 $r = A+B-t$ ;
            1.1.1 // Set  $POINTER[r]$  to the 1st element of  $VAR\_CANDIDATESTRIP[r]$ 
                 $POINTER[r] = VAR\_CANDIDATESTRIP[r]$ ;
            1.1.2 // Delete 1st element of  $VAR\_CANDIDATESTRIP[r]$  if it is obsolete
                 $GetFirst(r)$ ;
                if  $(Tmax \leq B)$  then  $DeleteFirst(r)$ ;
            1.2. // Processing of new candidates
                for all base pairs  $(A, B)$  from  $ROW_B$  in ascending order by  $A$  do begin
                    for all  $t$  from 0 to  $width-1$  do begin
                         $r = A+B-t$ ;
                    1.2.1  $IncludeCandidate(A, B, r)$ ;
                    end
                end

            2. // Work out sets  $\{(r-B, B-t) \mid t = 1, \dots, width-1\}$ ,
                // see (2.1), Supplementary section 3
            2.1. // Pre-processing of the 1st elements of old candidate lists
                for  $t=0; t < width; t++$  do begin
                     $B1 = B-t$ ;
                    for all base pairs  $(A, B1)$  from  $ROW_{B1}$  in descending order by  $A$  do begin
                         $r = A+B1$ ;
                    2.1.1 // Set  $POINTER[r]$  to the 1st element of  $VAR\_CANDIDATESTRIP[r]$ 
                         $POINTER[r] = VAR\_CANDIDATESTRIP[r]$ ;
                    2.1.2 // Delete 1st element of  $VAR\_CANDIDATESTRIP[r]$  if it is obsolete
                         $GetFirst(r)$ ;
                        if  $(Tmax \leq B)$  then  $DeleteFirst(r)$ ;
                    2.2. // Processing of new candidates
                        for  $t=0; t < width; t++$  do begin
                             $B1 = B-t$ ;
                            for all base pairs  $(A, B1)$  from  $ROW_{B1}$  in descending order by  $A$  do begin
                                 $r = A+B1$ ;
                            2.2.1  $IncludeCandidate(A, B, r)$ ;
                            end
                        end
                    end
                end
            end

```

Algorithm S4. Function *UpdateCandidate*.

```

algorithm IncludeCandidate(input:  $u, v, r, B$ )
// Tries to include the pair  $(u, v)$  to  $VAR\_CANDIDATESTRIP[r]$  within update
// of the list  $VAR\_CANDIDATESTRIP[r]$  from  $Cand_{r,B}$  to  $Cand_{r,B+1}$ .
// See section "Algorithm:3.Function InternalLoopStripRow and candidate lists".

var global
  array  $VAR\_CANDIDATESTRIP[2*L]$  of list of  $DAT\_CANDIDATE$ ;
  array  $POINTER[2*L]$  of pointers to list of  $DAT\_CANDIDATE$ ;

var local
   $DAT\_CANDIDATE$   $C$ ; // an element of  $VAR\_CANDIDATESTRIP[r]$ ,
                      // its components are  $C.x, C.y, C.dG, C.Tmin, C.Tmax$ 
  integer  $NEWmin, NEWmax$ ;
  float  $G, E1, G1$ ;

begin
   $G = \Delta G_r(u, v)$ ; // See (4), sub-section 2.1

  // 1. Find putative successor of  $(u, v)$  in  $VAR\_CANDIDATESTRIP[r]$ ,
  // i.e. the first element  $C$  with  $v-u \geq C.y - C.x$ ; see Lemma 3.2, Supplementary
  section 4.
  SetPointerToNext( $u, v, r$ );

  // 2. Kill obsolete successors
  while ( $POINTER[r] \neq NIL$ ) do begin
     $C = GetCurrent(r)$ ;
     $E1 = C.dG + f_{Len}(2*C.Tmax - r) - (C.y - C.x + 2)$ ;
     $G1 = G + f_{Len}(2*C.Tmax - r) - (v - u + 2)$ ;
    if ( $G1 \leq E1$ ) then
      DeleteCurrent( $r$ ); // POINTER[r] moves to the next element
    else break;
  end;

  // 3. Kill obsolete predecessors
   $POINTER[r] = Prev(POINTER[r])$ ;
  while ( $POINTER[r] \neq NIL$ ) do begin
     $C = GET\_CURRENT(r)$ ;
     $E1 = C.dG + f_{Len}(2*C.TMin - r) - (C.y - C.x + 2)$ ;
     $G1 = G + f_{Len}(2*C.TMin - r) - (v - u + 2)$ ;
    if ( $G1 \leq E1$ ) then
      DeleteCurrent( $r$ ); // POINTER[r] moves to the next element
       $POINTER[r] = Prev(POINTER[r])$ ;
    else break;
  end;

  // 4. Calculate NEWmin i.e. Tmin for  $(u,v)$ 
  // POINTER[r] points to the predecessor of  $(u, v)$ 
   $C = GetCurrent(r)$ ;
  if ( $C = NIL$ ) then  $NEWmin = B+1$  else begin
4.1.  $NEWmin = \min\{C.Tmax+1, \{t \in [C.Tmin, C.Tmax] \mid G + f_{Len}(2*t-r) - (u-v+2) >$ 
       $> C.dG + f_{Len}(2*t-r) - (C.y - C.x + 2)\}$ 
      }
    if ( $NEWmin \leq C.Tmax$ ) then begin
      SetCurrentTmax( $r, NEWmin-1$ );
    end
  end

  // 5. Calculate NEWmax i.e. Tmax for  $(u,v)$ 
   $POINTER[r] = Next(POINTER[r])$ ;
   $C = GetCurrent(r)$ ;
  if ( $C = NIL$ ) then  $NEWmax = L$  else begin
5.1.  $NEWmax = \max\{C.Tmin-1, \{t \in [C.Tmin, C.Tmax] \mid G + f_{Len}(2*t-r) - (u-v+2) >$ 
       $> C.dG + f_{Len}(2*t-r) - (C.y - C.x + 2)\}$ 
      }
  end

```

```

    if ( $NEWmax \geq C.Tmin$ ) then begin
      SetCurrentTmin( $r, NEWmax+1$ );
    end
end

    // 6. Insert ( $u, v, G$ )
    if ( $DN-1 < DP+1$ ) then STOP ;
    InsertBeforeCurrent( $r, u, v, G, DP+1, DN-1$ );
end.

```

Algorithm S5. Function *IncludeCandidate*. The function *SetPointerToNext*(u, v, r) moves *Pointer*[r] to the first element C in *VAR_CANDIDATESTRIP*[r] with $v-u \geq C.y - C.x$; see Lemma 3.2, Supplementary section 4. The function *GetCurrent*(r) extracts the element *Current*(r) of *VAR_CANDIDATESTRIP*[r], i.e. one corresponding to the value of *POINTER*[r]. The function *DeleteCurrent*(r) deletes the element *Current*(r) from the list *VAR_CANDIDATESTRIP*[r] and moves *POINTER*[r] to the next element. Functions *SetCurrentTmin* and *SetCurrentTmax* change the values *.Tmin* and *.Tmax* of *Current*(r). Note that in 4.1 $NEWmin > C.Tmin$, otherwise the element C would be deleted at step 3. Analogously because of step 2, $NEWmax < C.Tmax$ in line 5.1

Boolean function *SetPointerToNext*(input: u, v, r, B)
 // Finds putative predecessor of (u, v) in $VAR_CANDIDATESTRIP[r]$,
 // i.e. the last element C with $v-u \leq C.y - C.x$; see Lemma 3.2, Supplementary section 4.

```

var global
  array  $VAR\_CANDIDATESTRIP[2*L]$  of list of  $DAT\_CANDIDATE$ ;
  array  $POINTER[2*L]$  of pointers to list of  $DAT\_CANDIDATE$ ;
var local
   $DAT\_CANDIDATE$   $C$ ; // elements of  $VAR\_CANDIDATESTRIP[r]$ 
                       // its components are  $C.x, C.y, C.dG, C.Tmin, C.Tmax$ 
  float  $G$ ;
begin
   $G = \Delta G_i(u, v)$ ; // // See (4), section “Algorithm:1.Finding internal loops during
                       // construction of the optimal RNA structure”
   $C = GetCurrent(r)$ ;
  while ( $v-u < C.y - C.x$ ) do begin
    if ( $G \geq C.dG$ ) then return (FALSE);
     $POINTER[r] = Next(POINTER[r])$ ;
    if ( $POINTER[r] = NIL$ ) then return (TRUE);
    else  $C = GetCurrent(r)$ ;
  end
  if ( $v-u = C.y - C.x$  &  $G = C.E$ ) then return (FALSE);

```

Algorithm S6. Function *SetPointerToNext*; the candidate lists are implemented as ordinary lists. The function returns FALSE if the pairing $(u, v) \notin Cand_{r,B+1}$ because of condition 1.1 of Lemma 3.2, see Supplementary section 4.

Section 1. Sparse dynamic programming algorithms for the functions *InternalLoopMainRow* and *InternalLoopStripRow*

Calculation of $\Delta G_{Main}(A, B)$: general description of the algorithm

To calculate the values $\Delta G_{Main}(A, B)$ for all pairs $(A, B) \in ROW_B$ we use the divide and conquer algorithm. Let $Z(p, q)$, where $1 \leq p \leq q \leq L$, denote the set of base pairs:

$$Z(p, q) = ROW_p \cup \dots \cup ROW_q = \{(x, y) \mid (x, y) \in U \ \& \ p \leq y \leq q\}$$

The core problem to be solved (with proper input data) on each step of the algorithm is the following

Problem S2. Input:

- 1) integers p, k, q ; $1 \leq p < k \leq q \leq L$;
- 2) two sets of base pairs $P \subseteq Z(p, k-1)$ and $Q \subseteq Z(k, q)$;
- 3) real weights $F(x, y)$ for all $(x, y) \in P$.

Goal: for all $(A, B) \in Q$, to calculate:

$$F_{PARTIAL}(A, B; P) = \min\{F(x, y) + f_{Len}((B-A) - (y-x+2)) \mid (x, y) \in P\} \quad (1.1)$$

The algorithm referred to as *E-algorithm* below is a modification of the algorithm presented in Eppstein *et al.* (1992), section 3. The only difference between the E-algorithm and the original algorithm is an explicit description of forward and backward zones, introduced to fit the framework of the *OptimalRNA* algorithm, see Supplementary, algorithm S1.

On the pre-processing step of the main algorithm *OptimalRNA*, see Supplementary, algorithm S1, line 1, we assign to each row B integers p_B and q_B , where $p_B < B \leq q_B$. Sets $Z(p_B, B-1)$ and $Z(B, q_B)$ will be referred to as the *backward* and *forward zones* of B, respectively, and will be denoted as $BACKWARD_B$ and $FORWARD_B$. Informally, the set of zones corresponds to the execution of the divide-and-conquer algorithm of Eppstein *et al.* (1992) and can be represented as 2-3 tree with $\sim \log L$ levels, in which a zone of k -th level contains no more than $M/2^k$ base pairs.

For $q, B \in [1, L]$, $q > B$ let

$$WORKEDZONES(q; B) = \cup \{BACKWARD_k \mid k \leq B < q \leq q_k\}$$

In other words, $WORKEDZONES(q; B)$ is a union of all backward zones $BACKWARD_k$ of all rows k below B , their forward zones $FORWARD_k$ contain ROW_q .

Statement S1. There is the algorithm of zone assignment meeting the following conditions:

Z1. Each row ROW_B belongs to $O(\log L)$ backward and forward zones.

Z2. $WORKEDZONES(B; B) = Z(1, B-1)$ for all B .

Proof. The desired algorithm follows from the divide-and-conquer algorithm from Eppstein *et al.* (1992) and is presented in the Supplementary section 1.

Implementation of InternalLoopMainRow(B)

Let $P \subseteq U$ and $y < B$ for all $(x, y) \in P$ and

$$\Delta G_{MainPARTIAL}(A, B; P) = \min\{\Delta G(x, y) + f_{Len}((B-A) - (y-x+2)) \mid (x, y) \in P\} \quad (1.2)$$

To store the intermediate values of $\Delta G_{Main}(p, q)$ the function *InternalLoopMainRow* uses global variables $VAR_MAINWORK[x, y]$ assigned to each base pair $(x, y) \in U$ (see agreement on names in the **Introduction** section). The variables are initialized with $+\infty$ during the pre-processing step of the main algorithm *OptimalRNA*, see Supplementary, algorithm S1, line 1. After the call *InternalLoopMainRow(B)* the variable $VAR_MAINWORK[x, y]$ equals

$$\Delta G_{MainPARTIAL}(x, y; WORKEDZONES(y; B)) \quad (1.3)$$

According to the statement Z2, after calls *InternalLoopMainRow(k)*, $k = 1, \dots, B$, we thus will obtain for each $(A, B) \in ROW_B$,

$$\begin{aligned} VAR_MAINWORK[A, B] &= \Delta G_{MainPARTIAL}(A, B; WORKEDZONES(B; B)) = \\ &= \Delta G_{MainPARTIAL}(A, B; Z(1, B-1)) = \\ &= \Delta G_{Main}(A, B) \end{aligned} \quad (1.4)$$

Therefore, after all L calls *InternalLoopMainRow(B)*, $B \in [1, L]$, all variables $VAR_MAINWORK[x, y]$, where $(x, y) \in U$ will contain the desired values $\Delta G_{Main}(x, y)$.

To provide (1.3) and therefore (1.4), during the call *InternalLoopMainRow(B)* we solve the problem S2 with input data $p_B < B < q_B$, $P = BACKWARD_B$, $Q = FORWARD_B$; and weights $\Delta G(x, y)$. According to (1.1) and (1.2), for all base pairs (p, q) from $Z(B, q_B)$ we obtain the values $\Delta G_{MainPARTIAL}(p, q; BACKWARD_B)$.

Then we update the values of $VAR_MAINWORK[p, q]$ for all $(p, q) \in Z(B, q_B)$ by setting

$$\begin{aligned}
\text{VAR_MAINWORK } [p, q] &:= \\
&:= \min\{\Delta G_{\text{MainPARTIAL}}(p, q; \text{BACKWARD}_B), \text{VAR_MAINWORK } [p, q]\} = \\
&= \min\{\Delta G_{\text{MainPARTIAL}}(p, q; \text{BACKWARD}_B), \\
&\quad \Delta G_{\text{MainPARTIAL}}(p, q; \text{WorkedZones}(q; B-1))\} = \\
&= \Delta G_{\text{MainPARTIAL}}(p, q; \text{WorkedZones}(q; B))
\end{aligned}$$

Solution of the problem S2

The algorithm *ECore* for solving problem S2 is based on the Dynamic minimization procedure, described in the section 2 of Eppstein *et al.* (1992). Its run-time is $O(N*\log N)$, where N is the total number of base pairs in $P \cup Q$, and the needed work space is $O(N)$. *ECore* looks over all base pairs from $P \cup Q$ in descending order of their x -coordinates, i.e. lower elements of base pairs. At a moment $t \in [1, L]$ *ECore* stores the set

$$\text{ACTIVE}_t \subseteq \{(x, y) \mid (x, y) \in P, x \geq t\},$$

which consists of all base pairs (x, y) from $\{(x, y) \in P, x \geq t\}$ that can provide minimum in (1.1) for some base pair $(p, q) \in Q$ with $p < t$.

Since f_{Len} is convex, the modification of the set ACTIVE_t (if the current base pair belongs to P) and search in ACTIVE_t (if the current base pair belongs to Q) can be performed on average in $\log N$ operations. The description of the data structure DAT_ACTIVE , used to store the current set ACTIVE_t and details of Dynamic minimization procedure can be found in Eppstein *et al.* (1992), section 2.

The run-time of call *InternalLoopMainRow*(B) is determined by the run-time of the procedure *ECore* and is $O(N*\log N)$, where N is a total number of base pairs in backward and forward zones of the B . Thus, (see statement Z1) the total time of all L calls of *InternalLoopMainRow* and the initialization is $O(M*\log^2 L)$.

Calculation of $\Delta G_{Strip}(A, B)$

The *ES-algorithm* implements the function `InternalLoopStripRow`, see Supplementary, algorithm S2 i.e. calculates values $\Delta G_{Strip}(A, B)$ for all pairs $(A, B) \in U$. The algorithm is given below. Analogously to the *E-algorithm* it exploits SDP. Subroutines *GetMinimumFromActive* and *UpdateActive* follows Eppstein *et al.* (1992).

```

Function InternalLoopStripRow_ES(input: B){
// Computes values  $\Delta G_{Strip}(A, B)$  for all  $(A, B) \in ROW_B$ ;
// Updates variables  $VAR\_STRIPWORK[A, B]$  for all  $(A, B) \in FORWARD_B$ ;
data
    array  $VAR\_ACTIVESTRIP[L]$  of  $DAT\_ACTIVE$ ;
begin
    // 1. Pre-processing
    1.1. // Initialize  $VAR\_ACTIVESTRIP$ 
        for all base pairs  $(x, y) \in BACKWARD_B \cup FORWARD_B$ 
            in descendant order by x, then in ascendant order by y do begin
                 $r = x+y$ ;
                if  $(x, y) \in FORWARD_B$  then begin
                     $VAR\_ACTIVESTRIP[r-B] := empty$ ;
                end
                if  $(x, y) \in BACKWARD_B$  then
                    for t from  $\max\{t-width+1, B+1\}$  to  $\min\{t+width-1, 2*B-1\}$  do begin
                         $VAR\_ACTIVESTRIP[t-B] := empty$ 
                    end
                end
            end
        // Main loop
    2. for all base pairs  $(x, y) \in BACKWARD_B \cup FORWARD_B$ 
        in descendant order by x, then in ascendant order by y do begin
    3.      $r = x+y$ ;
    4.     if  $(x, y) \in FORWARD_B$  then begin
             $TEMP = GetMinimumFromActive(x, y, VAR\_ACTIVESTRIP[r-B])$ ;
             $VAR\_STRIPWORK[x, y] = \min\{TEMP, VAR\_STRIPWORK[x, y]\}$ ;
        end
    5.     if  $(x, y) \in BACKWARD_B$  then
            for t from  $\max\{t-width+1, B+1\}$  to  $\min\{t+width-1, 2*B-1\}$  do begin
                 $UpdateActive(x, y, VAR\_ACTIVESTRIP[r-B])$ ;
            end
        end
    end
end

```

During the call *InternalLoopStripRow*(B), the *ES-algorithm* solves several copies of the Problem S2, each corresponding to a diagonal $DIAG_r$, $r = B+1, \dots, 2*B-1$. For all copies the integer input values are p_B , B and q_B . Thus, we use the same zone assignment as for *InternalLoopMainRow* function. The input sets for the copy corresponding to a diagonal r are $P = BACKWARD_B \cap STRIP_r$; $Q = FORWARD_B \cap DIAG_r$; and the weight function

$F(x,y) = \Delta G_r(x, y)$, see (4). For each base pair $(x, y) \in U$ we have a variable $VAR_STRIPWORK[x, y]$ which is used analogously to the variable $VAR_MAINWORK[x, y]$ of the previous section.

During the call $InternalLoopStripRow(B)$, all $B-1$ copies of the Problem S2 are solved in a parallel way. To do this we use the array $VAR_ACTIVESTRIP$ of length L , its elements are DAT_ACTIVE data structure (see sub-section “**Solution of the problem S2**”). The element $VAR_ACTIVESTRIP[i]$ contains current value of the set $ACTIVE$ for the copy of the Problem S2 corresponding to the diagonal $DIAG_{B+i}$. For a base pair $(x, y) \in BACKWARD_B$ we first determine the range of r that $(x, y) \in STRIP_r$, then we modify $VAR_ACTIVESTRIP[r-B]$ for all r from the range. For a base pair $(p, q) \in FORWARD_B$, we look for a desired minimum in $VAR_ACTIVESTRIP[p+q-B]$.

Each $(p, q) \in BACKWARD_B$ belongs to at most $2*width-1$ strips. Thus, total run time of all calls of $InternalLoopStripRow$ is at most $2*width-1$ times larger than the run time of $InternalLoopMainRow$. Therefore, the run-time of all calls of $InternalLoopStripRow$ is $O(width*M*\log^2L)$.

Section 2. Proof of Statement S1.

Statement S1. There is the algorithm of zone assignment meeting the following conditions:

Z1. Each row ROW_B belongs to $O(\log L)$ backward and forward zones.

Z2. $WORKEDZONES(B; B) = Z(1, B-1)$ for all B .

Proof. The algorithm follows the divide-and-conquer procedure from Eppstein *et al.* (1992). The zone tree is a rooted tree meeting following conditions.

T1. The tree is 2-3 tree, i.e. each its inner node has 2 or 3 sons, the sons are ordered. The i -th son of a node V will be denoted as $Son(V, i)$.

T2. Each node V of the tree is assigned with a range of rows $Range(V) = [s(V), t(V)]$; here $s(V)$ and $t(V)$ are the first and the last row of the range; $1 \leq s(V) \leq t(V) \leq L$. $Size(V)$ denotes the size of the zone $Z(s(V), t(V))$, i.e. the total number of allowed base pairs in rows $ROW_{s(V)}, \dots, ROW_{t(V)}$.

T3. If R is a root of the tree, then $Range(R) = [1, L] = U$.

T4. Let V be an inner node of the tree. Then only one of 4 following relations between V and its sons are possible:

4.1. V has two sons V_1 and V_2 and

$$a) s(V_1) = s(V); s(V_2) = t(V_1)+1; t(V_2) = t(V);$$

b) $Size(V)$ is even and $Size(V_1) = Size(V_2) = Size(V)/2$.

4.2. V has two sons V_1 and V_2 and

a) $t(V_1) = s(V_1) = s(V)$; $s(V_2) = s(V)+1 = t(V_1)+1$; $t(V_2) = t(V)$;

b) $Size(V_2) < Size(V)/2$.

4.3. V has two sons V_1 and V_2 and

a) $s(V_1) = s(V)$; $t(V_1) = t(V)-1$; $s(V_2) = t(V_1)+1 = t(V_2) = t(V)$;

b) $Size(V_1) < Size(V)/2$.

4.4. V has three sons V_1 , V_2 , and V_3 and

a) $s(V_1) = s(V)$; $t(V_2) = s(V_2) = t(V_1)+1$; $s(V_3) = t(V_2)+1$; $t(V_3) = t(V)$;

b) $Size(V_1) < Size(V)/2$; $Size(V_2) < Size(V)/2$.

T5. If V is a leaf, then its zone contains exactly one non-empty row.

According Eppstein *et al.* (1992) we will define the zone tree for the given set U of base pairings by the following induction.

E0. The initial tree T_0 consists of the root R assigned with the zone $Z(1, L)$.

E1. Let we have a tree T_k meeting conditions T1–T4, and V is the first (according to the lexicographic order corresponding to the ordering of sons) leaf of T_k that does not fit T5; $Range(V) = [s(V), t(V)]$. The tree T_{k+1} will be obtain by addition the sons of V according the following rules.

E.1.1 Let $Size(Z(s(V), s(V))) > Size(V)/2$. Then we add two sons V_1 and V_2 and set $s(V_1) = t(V_1) = s(V)$; $s(V_2) = s(V)+1$; $t(V_2) = t(V)$. The nodes V , V_1 and V_2 correspond to the case 4.2.

E.1.2. Let $Size(Z(s(V), s(V))) \leq Size(V)/2$ and let

$$b = \max\{d \leq t(V) \mid Size(Z(s(V), d)) \leq Size(V)/2\}.$$

Obviously, $1 \leq b \leq t(V)-1$. If $Size(Z(s(V), b)) = Size(V)/2$, then we add two sons V_1 and V_2 and set $s(V_1) = s(V)$; $t(V_1) = b$; $s(V_2) = b+1$; $t(V_2) = t(V)$. This corresponds to the case 4.1.

E.1.3. Let $Size(Z(s(V), b)) < Size(V)/2$ and $b = t(V)-1$. We add two sons V_1 and V_2 and set $s(V_1) = s(V)$; $t(V_1) = b = t(V)-1$; $s(V_2) = b+1 = t(V)$; $t(V_2) = t(V)$; this corresponds to the case 4.3.

E.1.4. Finally, let $Size(Z(s(V), b)) < Size(V)/2$ and $b \leq t(V)-2$. This implies $Size(Z(t(V), t(V))) < Size(V)/2$. In this case we add three new nodes V_1 , V_2 , V_3 and assign them with zones as follows:

$$s(V_1) = s(V); t(V_1) = b; t(V_2) = s(V_2) = b+1; s(V_3) = t(V_2)+1 = b+2; t(V_3) = t(V).$$

Let's show that this corresponds to the case 4.4. Indeed, if

$$Size(V_3) = Size(Z(b+2, t(V))) \geq Size(V)/2$$

then $Size(Z(s(V), b+1)) \leq Size(V)/2$ that contradicts the definition of b .

Lemma 1.1.

1. The described procedure converges to the zone tree $ZT(U)$.
2. The height of the obtained zone tree $ZT(U)$ does not exceed $\log M$.

Proof. Let V be a node of an intermediate tree T_r , the level of V is k , and the range $Range(V)$ contains at least two non-empty rows. One can see that in this case $Size(V) \leq M/2^k$. The lemma directly follows from this observation.

Lemma 1.2

1. Let $B \in [1, L]$ and the row ROW_B is not-empty. Then there is exactly one leaf V of the tree $ZT(U)$ such that $B \in Range(V)$. If V is the leaf of $ZT(U)$ then there is exactly one non-empty row B such that $B \in Range(V)$. This defines the one-to-one correspondence between non-empty rows in U and leaves of $ZT(U)$.

The leaf corresponding to the non-empty row B will be denoted as $Leaf(B)$.

2. Let B, B' are non-empty rows, $B < B'$. Then $Leaf(B)$ precedes $Leaf(B')$ in lexicographic order of the nodes of $ZT(U)$.

3. Let V is the leaf corresponding to the non-empty row B and $V_0 = R, V_1, \dots, V_n = V$ are all nodes on the path from the root to V . Then for a node W of V

$$B \in Range(W) \Leftrightarrow W \in \{ V_0, V_1, \dots, V_n \}$$

Proof. The next lemma follows from the definition of the zone tree. For the sake of brevity we do not give its proof here.

We say that the node V is *weak* if it is a root or is the 1st son of its parent.

Otherwise, the node is *strong*.

Lemma 1.3.

1. Let B is non-empty row and it is not the first non-empty row, i.e. there is a non-empty row b , where $b < B$. Then the path from the root R to the leaf $Leaf(B)$ contains at least one strong node.

2. Let V be the last strong node on the path from the root R to the leaf $Leaf(B)$. Then B is the first non-empty row in the $Range(V)$.

Proof. Follows from the claim 2 of Lemma 1.2.

Let B is the non-first non-empty row. The last strong node on the path from the root to the $Leaf(B)$ will be denoted as $Strong(B)$.

Now we are ready to describe the algorithm assigning zone borders p_B, q_B to all non-empty rows $ROW_B, B \in [1, L]$, except of the first non-empty row. The algorithm consists of two steps.

A1. Create the zone tree $ZT(U)$ according E0, E1.

A2. Let B be the non-first non-empty row; $V = Strong(B)$ and W is a Parent of V .

Then $p_B = s(W); q_B = t(V)$.

Obviously, the run-time of the algorithm is $O(L \cdot \log L)$. The claim Z1 follows from the claim 2 of Lemma 1.1 and the claim 3 of lemma 1.2. Consider the claim Z2. Let B be the non-first non-empty row. According to A2 $B \in FORWARD_x \Leftrightarrow B \in Range(Strong(x))$.

According to Lemma 1.2, claim 3, this implies

$$B \in FORWARD_x \Leftrightarrow Strong(x) \text{ belongs to the path from the root to } Strong(B) \quad (2.1)$$

The claim Z2 can be proven by induction on the number of strong nodes on the path from the root to $Strong(B)$. Suppose, that the path does not contain strong nodes. Then all these nodes are weak and thus for the farther of $Strong(B)$ we have $s(W) = 1$. Then $p_B = 1$ and $BACKWARD_B = [1, B-1]$. Let $Strong(B')$ is the last strong node the path from the root to $Strong(B)$. Obviously, $B' < B$ and by induction

$$WORKEDZONES(B'; B') = Z(1, B'-1) \quad (2.2)$$

According to (2.1) and because of definition of B' ,

$$WORKEDZONES(B'; B') \in WORKEDZONES(B; B) \quad (2.3)$$

Analogously to the basic of induction we can show that

$$BACKWARD_B = Z(s(Strong(B')), B-1) \quad (2.4)$$

According to the Lemma 1.3, claim 2,

$$Z(s(Strong(B')), B-1) = Z(B', B-1) \quad (2.5)$$

Now the claim Z2 follows from (2.2) – (2.5)

Section 3. Proof of Statement 1.

Statement 1. Let $(x_1, y_1), \dots, (x_n, y_n)$ be all (r, B) -candidates, ordered by decrement of $d_i = y_i - x_i$. Then,

1. All values $y_1 - x_1, \dots, y_n - x_n$ are different, i. e. $y_1 - x_1 > \dots > y_n - x_n$;
2. $\Delta G_r(x_1, y_1) > \dots > \Delta G_r(x_n, y_n)$;
3. There are values $T_0 = B - 1 < T_1 < \dots < T_{n-1} < T_n = L$ such that $HOME_{r,B}(x_i, y_i) = [T_{i-1} + 1, T_i]$.

Proof.

1 Suppose that $y_i - x_i = y_{i+1} - x_{i+1}$. If, e.g. $\Delta G_r(x_i, y_i) < \Delta G_r(x_{i+1}, y_{i+1})$, then for all $(p, q) \in DIAG_r$

$$\Delta G_r(x_i, y_i) + f_{Len}((q-p) - (y_i - x_i + 2)) < \Delta G_r(x_{i+1}, y_{i+1}) + f_{Len}((q-p) - (y_{i+1} - x_{i+1} + 2)), \text{ or}$$

$$\Delta G_{IStruct}(x_i, y_i; p, q) < \Delta G_{IStruct}(x_{i+1}, y_{i+1}; p, q),$$

and (x_{i+1}, y_{i+1}) cannot be a (r, B) -candidate. Then, $\Delta G_r(x_i, y_i) = \Delta G_r(x_{i+1}, y_{i+1})$. In this case, according to (c) of the definition of $OWNER_{r,B}(q, p)$, only one of pairs $(x_i, y_i), (x_{i+1}, y_{i+1})$ can be a (r, B) -candidate. The contradiction proves the statement.

2. By definition, for all $i < n$ we have: $y_i - x_i > y_{i+1} - x_{i+1}$, and thus for any $(p, q) \in DIAG_r, p+q = r, q \geq B$

$$(q-p) - (y_i - x_i + 2) < (q-p) - (y_{i+1} - x_{i+1} + 2)$$

Function f_{Len} increases monotonously, thus

$$f_{Len}((q-p) - (y_i - x_i + 2)) < f_{Len}((q-p) - (y_{i+1} - x_{i+1} + 2)) \quad (3.1)$$

Now one can see that $\Delta G_r(x_i, y_i) \leq \Delta G_r(x_{i+1}, y_{i+1})$ together with (3.1) implies that for all $(p, q) \in DIAG_r, p+q = r, q \geq B$

$$\Delta G_r(x_i, y_i) + f_{Len}((q-p) - (y_i - x_i + 2)) < \Delta G_r(x_{i+1}, y_{i+1}) + f_{Len}((q-p) - (y_{i+1} - x_{i+1} + 2))$$

The latter is impossible, because (x_{i+1}, y_{i+1}) is the (r, B) -candidate.

3. The statement utilizes the convexity of the function f_{Len} and follows from the lemma given below (see also Eppstein *et al.* (1992)).

Lemma 2.1. Let $(A, B), (A', B') \in U; A+B = A'+B' = r$; and $B' > B$.

Let $(x, y), (x', y') \in U(A, B) \subset U(A', B')$, and $y-x < y'-x'$ and

$$\Delta G_r(x, y) + f_{Len}((B-A) - (y-x+2)) < \Delta G_r(x', y') + f_{Len}((B-A) - (y'-x'+2)) \quad (3.2)$$

Then

$$\Delta G_r(x, y) + f_{Len}((B' - A') - (y - x + 2)) < \Delta G_r(x', y') + f_{Len}((B' - A') - (y' - x' + 2))$$

Proof. We have to prove that

$$f_{Len}((B' - A') - (y - x + 2)) - f_{Len}((B' - A') - (y' - x' + 2)) < \Delta G_r(x', y') - \Delta G_r(x, y)$$

According to (3.2) it is enough to prove that

$$\begin{aligned} f_{Len}((B' - A') - (y - x + 2)) - f_{Len}((B' - A') - (y' - x' + 2)) < \\ < f_{Len}((B - A) - (y - x + 2)) - f_{Len}((B - A) - (y' - x' + 2)) \end{aligned}$$

or, equivalently,

$$\begin{aligned} f_{Len}((B' - A') - (y - x + 2)) + f_{Len}((B - A) - (y' - x' + 2)) < \\ < f_{Len}((B' - A') - (y' - x' + 2)) + f_{Len}((B - A) - (y - x + 2)) \end{aligned} \quad (3.3)$$

Note, that

$$\begin{aligned} [(B' - A') - (y - x + 2)] + [(B - A) - (y' - x' + 2)] = \\ [(B' - A') - (y' - x' + 2)] + [(B - A) - (y - x + 2)] \end{aligned} \quad (3.4)$$

and

$$0 < [(B - A) - (y' - x' + 2)] < [(B' - A') - (y' - x' + 2)] < [(B' - A') - (y - x + 2)] \quad (3.5)$$

$$0 < [(B - A) - (y' - x' + 2)] < [(B - A) - (y - x + 2)] < [(B' - A') - (y - x + 2)] \quad (3.6)$$

Now (3.3) follows from convexity of the function f_{Len} together with (3.4), (3.5) and (3.6).

Section 4. Proof of Statement 2, part 1.

Statement 2.

1. The function $InternalLoopStripRow_G(B)$ correctly calculates values $\Delta G_{Strip}(A, B)$ within the algorithm $OptimalRNA_G$.
2. The total run-time of calls $InternalLoopStripRow_G(B)$ and $UpdateCandidate(B)$ within the work of the algorithm $OptimalRNA_G$ (see Supplementary, algorithms S3–S5) $O(width * M * \log L)$, $f_{Len}(x)$ is supposed to be a convex function, its the value can be calculated in constant time.
3. If $f_{Len}(x)$ is logarithmic function, then the total run-time of calls $InternalLoopStripRow_G(B)$ and $UpdateCandidate(B)$ within the work of the algorithm $OptimalRNA_G$ (see Supplementary, algorithms S3–S5) is $O(width^2 * M)$, the value $f_{Len}(x)$ is supposed to be calculated in constant time.
4. The workspace of the $OptimalRNA_G$ is $O(L) + O(width * M)$

Proof. 1. The implementation of $InternalLoopStripRow_G(B)$ is shown in Supplementary, algorithm S3 Suppose, that before the call $InternalLoopStripRow_G(B)$ the variable $VAR_CANDIDATESTRIP[r]$ contains the candidate list $CAND_{r,B}$. Consider the pairing $(A, B) \in ROW_B$ and let $r = A+B$. The function $GetFirst(r)$ gets the 1st element $C = (C.x, C.y, C.dG, C.Tmin, C.Tmax)$ of the list $VAR_CANDIDATESTRIP[r] = CAND_{r,B}$. According to statement 1, $(C.x, C.y) = OWNER_{r,B}(r-B, B) = OWNER_{A+B,B}(A, B)$. Thus, the value $\Delta G_{Strip}(A, B)$ can be calculated as

$$\Delta G_{Strip}(A, B) = \Delta G_{A+B}(C.x, C.y) + f_{Len}(((A+B) - 2*B) - (C.y - C.x + 2))$$

To finish the proof we have to show that $UpdateCandidate(B)$ correctly updates candidate lists. To update the list $VAR_CANDIDATESTRIP [r]$ from $CAND_{r,B}$ to $CAND_{r,B+1}$ one has to work up base pairs from the set

$$DELTA_{r,B} = STRIPPRED_{r(B+1)} \setminus STRIPPRED_r(B).$$

Note, that

$$DELTA_{r,B} \subseteq \{(r-B+t, B) | t = 0, 1, \dots, width-1\} \cup \{(r-B, B-t) | t = 1, \dots, width-1\}$$

We first work out all new candidates that belong to ROW_B (see line 1.1, Supplementary algorithm S4, then all other candidates (see line 1.2). This determines the ordering of all elements of $DELTA_{r,B}$: $(u_1, v_1), (u_2, v_2), \dots, (u_s, v_s)$. Let $D_f(r, B) = STRIPPRED_r(B) \cup \{(u_1, v_1), \dots, (u_j, v_j)\}$. Obviously,

$$D_0(r, B) = STRIPPRED_r(B);$$

$$D_s(r, B) = STRIPPRED_r(B+1).$$

Let $(p, q) \in DIAG_r$, $q \geq B+1$, $r = p+q$. Analogously to $OWNER_{r,B}(p, q) \in STRIPPRED_r(B)$, we define $TEMPOWNER_{r,B}(p, q, j) \in D_j(r, B)$ as a minimal element according the ordering: $(x, y) <_{\otimes} (x', y')$ defined in the sub-section “**2.3.Function InternalLoopStripRow and candidate lists**”. We say that $(x, y) \in D_j(r, B)$ is an (r, B, j) -candidate if $(x, y) = TEMPOWNER_{r,B}(p, q, j)$ for some $(p, q) \in U$, $q \geq B+1$.

Lemma 3.1. Statement 1 is valid for the lists of (r, B, j) -candidates. I. e. let $(x_1, y_1), \dots, (x_n, y_n)$ be all (r, B, j) -candidates, ordered by decrement of $d_i = y_i - x_i$. Then,

1.1. All values $y_1 - x_1, \dots, y_n - x_n$ are different, i. e. $y_1 - x_1 > \dots > y_n - x_n$;

1.2. $\Delta G_r(x_1, y_1) > \dots > \Delta G_r(x_n, y_n)$;

1.3. There are values $T_0 = B < T_1 < \dots < T_{n-1} < T_n = L$ such that $HOME_{r,B,j}(x_i, y_i) = [T_{i-1}+1, T_i]$, where (r, B, j) -home $HOME_{r,B,j}(x, y)$ of the (r, B, j) -candidate (x, y) is a set of all q such that $B \leq q < r$ and $(x, y) = OWNER_{r,B,j}(r-q, q)$.

Proof. is exactly the same as for Statement 1. See Supplementary section 3.

Let $CAND_{r,B,j-1} = \{C_i\}$, $i = 1, \dots, N$; $C_i = \langle C_i.x, C_i.y, C_i.dG, C_i.Tmin, C_i.Tmax \rangle$. Consider $j \in \{1, \dots, s\}$ and let $(u, v) = (u_j, v_j) \in DELTA_{r,B}$ and $G = \Delta G_r(u, v)$, see (4). Let further $C_d = \langle PREV.x, PREV.y, PREV.dG, PREV.Tmin, PREV.Tmax \rangle$ be the last element of $CAND_{r,B,j-1}$ with $PREV.y - PREV.x \geq v - u$. We set $d=0$ if $v - u > C_i.y - C_i.x$ for all $i = 1, \dots, N$. Let's define $Start(u, v; i)$, where $i \leq d$ as

$$Start(u, v; i) =$$

$$= \min \{ C_i.Tmax+1 \cup \dots \cup \{ t \in [C_i.Tmin, C_i.Tmax] \mid$$

$$\quad | G + f_{Len}((2*t - r) - (u-v+2)) < C_i.dG + f_{Len}((2*t - r) - (C_i.y - C_i.x + 2)) \}$$

$$\}$$

Analogously, for $i > d$ we define

$$End(u, v; i) =$$

$$= \max \{ C_i.Tmin-1 \cup \dots \cup \{ t \in [C_i.Tmin, C_i.Tmax] \mid$$

$$\quad | G + f_{Len}((2*t - r) - (u-v+2)) < C_i.dG + f_{Len}((2*t - r) - (C_i.y - C_i.x + 2)) \}$$

$$\}$$

Finally, let

$$Start(u, v) = Start(u, v; d), \text{ if } d \geq 1; \quad Start(u, v) = B+1 \text{ otherwise};$$

$$End(u, v) = End(u, v; d+1), \text{ if } d \leq N; \quad End(u, v) = L \text{ otherwise};$$

Lemma 3.2.

1. $(u, v) \in CAND_{r,B,j}$ if and only if

1.1. $G < PREV.dG$ and

1.2. $End(u, v; i) > Start(u, v; i)$

2. Let $i \leq d$. Then $C_i \in CAND_{r,B,j}$ if and only if $G_{pred}(i) > E_{pred}(i)$, where

$$E_{pred}(i) = C_i.dG + f_{Len}(2 * C_i.Tmin - r) - (C_i.y - C_i.x + 2);$$

$$G_{pred}(i) = G + f_{Len}(2 * C_i.Tmin - r) - (v - y + 2);$$

3. Let $i > d$. Then $C_i \in CAND_{r,B,j}$ if and only if $G_{succ}(i) > E_{succ}(i)$, where

$$E_{succ}(i) = C_i.dG + f_{Len}(2 * C_i.Tmax - r) - (C_i.y - C_i.x + 2);$$

$$G_{succ}(i) = G + f_{Len}(2 * C_i.Tmax - r) - (v - y + 2);$$

4. Let $(u, v) \in CAND_{r,B,j}$. and $C_i \in CAND_{r,B,j-1}$ is the immediate predecessor of (u, v) in $CAND_{r,B,j}$. Then the value $Tmin$ corresponding to (u, v) can be found as

$$\begin{aligned} Tmin &= \\ &= \min\{t \mid G + f_{Len}((2 * t - r) - (u - v + 2)) \leq C_i.dG + f_{Len}((2 * t - r) - (C_i.y - C_i.x + 2))\} = \\ &= Start(u, v; i) \end{aligned}$$

5. Let $(u, v) \in CAND_{r,B,j}$. and $C_i \in CAND_{r,B,j-1}$ is the immediate successor of (u, v) in $CAND_{r,B,j}$. Then the value $Tmax$ corresponding to (u, v) can be found as

$$\begin{aligned} Tmax &= \\ &= \max\{t \mid G + f_{Len}((2 * t - r) - (u - v + 2)) < C_i.dG + f_{Len}((2 * t - r) - (C_i.y - C_i.x + 2))\} = \\ &= End(u, v; i) \end{aligned}$$

Proof. See Supplementary section 6.

The function *IncludeCandidate* directly follows Lemma 3.2, its steps correspond to the claims of the Lemma. The only difference is that we avoid checking condition 1.2 at step 1 and do this only at step 6. This completes the proof of the 1st part of the Statement 2.

Section 5. Proof of Statement 2, parts 2, 3.

Statement 2.

1. The function *InternalLoopStripRow_G(B)* correctly calculates values $\Delta G_{Strip}(A, B)$ within the algorithm *OptimalRNA_G*.

2. The total run-time of calls *InternalLoopStripRow_G(B)* and *UpdateCandidate(B)* within the work of the algorithm *OptimalRNA_G* (see Supplementary, algorithms S3–S5) $O(\text{width} * M * \log L)$, $f_{Len}(x)$ is supposed to be a convex function, its the value can be calculated in constant time.

3. If $f_{Len}(x)$ is logarithmic function, then the total run-time of calls *InternalLoopStripRow_G(B)* and *UpdateCandidate(B)* within the work of the algorithm *OptimalRNA_G* (see Supplementary, algorithms S3–S5) is $O(\text{width}^2 * M)$, the value $f_{Len}(x)$ is supposed to be calculated in constant time.

4. The workspace of the *OptimalRNA_G* is $O(L) + O(\text{width} * M)$

Proof. The total run time of all calls of *InternalLoopStripRow_G*, see Supplementary, algorithm S3 is $O(M)$, because the call for the given $(A, B) \in U$ the call *GetFirst(B, A+B)* and the computation of *VAR_STRIPWORK[A, B]* can be performed in a constant time.

The pre-processing steps of the function *UpdateCandidate*, see lines 1.1 and 2.1, Supplementary algorithm S4 are performed in $O(\text{width} * M)$ time, because each of call of lines 1.1.1, 1.1.2, 2.1.1, 2.1.2 is performed in constant time and each of these lines is called $O(\text{width} * M)$ times.

To accomplish the proof we have to consider the function *IncludeCandidate*, see Supplementary algorithm S5, its steps correspond to the claims of Lemma 3.2, see Supplementary section 4. Step 1 is the most tiresome one and we will consider it at the end of the section.

The total run-time $T_{2,3}$ of steps 2 and 3 during all calls of *IncludeCandidate* can be represented as

$$T_{2,3} = O(N_{Incl}) + O(N_{Del} * T_{Del})$$

where N_{Incl} is the number of calls of *IncludeCandidate*, N_{Del} is the number of calls of *DeleteCurrent*, and T_{Del} is the time needed to delete the current element of the candidate list. Each pairing $(x, y) \in U$ can be included only in $O(\text{width})$ lists, therefore $N_{Incl} =$

$O(\text{width} * M)$. In turn, each $(x, y) \in U$ can be deleted from each candidate list at most once. Thus, $N_{Del} = O(\text{width} * M)$. The time T_{Del} is $O(1)$ if we represent candidate lists as ordinary lists and $O(\log L)$ if tree-like structure is utilized (see below). Therefore we have

$$T_{2,3} = O(\text{width} * M * T_{Del}) \quad (5.1)$$

where

$$T_{Del} = O(1) \text{ or } T_{Del} = O(\log L) \quad (5.2)$$

Analogously, for the total run-time T_6 of steps 6 during all calls of *IncludeCandidate* we have

$$T_6 = O(N_{Incl} * T_{Ins}) = O(\text{width} * M * T_{Ins}) \quad (5.3)$$

where T_{Ins} is the time needed to insert the new element of the candidate list. Depending of the chosen implementation

$$T_{Ins} = O(1) \text{ or } T_{Ins} = O(\log L) \quad (5.4)$$

The total run-time $T_{4,5}$ of steps 4 and 5 during all calls of *IncludeCandidate* is $O(N_{Incl} * T_{Root})$, where T_{Root} is the time needed to find values of DP and DN , see lines 4.1 and 5.1. If $f_{Len}(x)$ is logarithmic function this can be done in constant time. In case of a general convex function, the binary search (see Lemma 2.1, Supplementary section 3) leads to $T_{Root} = O(\log L)$. Therefore, we again obtain two variants of formula for $T_{4,5}$:

$$T_{4,5} = O(\text{width} * M * \log L) \quad (5.5)$$

for the arbitrary convex function f_{Len} and

$$T_{4,5} = O(\text{width} * M) \quad (5.6)$$

if f_{Len} is logarithmic function.

Now we come to the step 1. The total run-time T_1 of this step 5 during all calls of *IncludeCandidate* can be represented as

$$T_1 = O(N_{Incl} * T_{Find}) = O(\text{width} * M * T_{Find}), \quad (5.7)$$

where T_{Find} is the average run-time of the function *SetPointerToPred*, i.e., needed to locate in the candidate list the putative predecessor of the given pairing (u, v) , see Lemma 3.2, Supplementary section 4. The function *SetPointerToPred* can be implemented in several ways. First, we can represent the lists as balanced trees (see, e.g. Aho *et al.*, 1974) and this gives us

$$T_{Find} = O(\log L).$$

In combination with (5.1), (5.3), (5.5) and (5.7) this gives us the desired estimation of the overall run-time. However, the other bounds are also possible. We can represent candidate lists as ordinary lists and obtain

$$T_{Find} = O(\text{width})$$

that with (5.7) and the above estimations $T_{2,3}$, $T_{4,5}$, and T_6 for gives estimation $O(\text{width}^2 * M)$ both for T_1 and for overall run-time. This proves the claim 3 of the statement 2.

The workspace of the algorithm *OptimalRNA_G* is determined by the space needed to store the candidate lists. Note, that each $(x, y) \in U$ can belong at most to $O(\text{width})$ lists. The claim 4 follows from the above observations.

Section 6. Proof of Lemma 3.2.

Let $DELTA_{r,B}: \{(u_1, v_1), (u_2, v_2), \dots, (u_s, v_s)\}$, see notation in the Supplementary section 4.

Consider $j \in \{1, \dots, s\}$. Let $CAND_{r,B,j-1} = \{C_i\}, i = 1, \dots, N; C_i = \langle C_{i,x}, C_{i,y}, C_i.dG, C_i.Tmin, C_i.Tmax \rangle$. Let further $(u, v) = (u_j, v_j) \in DELTA_{r,B}; G = \Delta G_r(u, v)$, see (4) and

$$C_d = \langle PREV.x, PREV.y, PREV.dG, PREV.Tmin, PREV.Tmax \rangle$$

be the last element of $CAND_{r,B,j-1}$ with $PREV.y - PREV.x \geq v - u$. We set $d = 0$ if $v - u > C_{i,y} - C_{i,x}$ for all $i = 1, \dots, N$. Let's define $Start(u, v; i)$, where $i \leq d$ as

$$\begin{aligned} Start(u, v; i) &= \\ &= \min \{ C_i.Tmax + 1 \cup \dots \cup \{ t \in [C_i.Tmin, C_i.Tmax] \mid \\ &\quad | G + f_{Len}((2*t - r) - (u - v + 2)) < C_i.dG + f_{Len}((2*t - r) - (C_{i,y} - C_{i,x} + 2)) \} \\ &\quad \} \end{aligned}$$

Analogously, for $i > d$ we define

$$\begin{aligned} End(u, v; i) &= \\ &= \max \{ C_i.Tmin - 1 \cup \dots \cup \{ t \in [C_i.Tmin, C_i.Tmax] \mid \\ &\quad | G + f_{Len}((2*t - r) - (u - v + 2)) < C_i.dG + f_{Len}((2*t - r) - (C_{i,y} - C_{i,x} + 2)) \} \\ &\quad \} \end{aligned}$$

Finally, let

$$\begin{aligned} Start(u, v) &= Start(u, v; d), \text{ if } d \geq 1; Start(u, v) = B + 1 \text{ otherwise;} \\ End(u, v) &= End(u, v; d + 1), \text{ if } d \leq N; End(u, v) = L \text{ otherwise;} \end{aligned}$$

Lemma 3.2.

1. Let $(u, v) \in CAND_{r,B,j}$. Then $G < PREV.dG$ and
2. $(u, v) \in CAND_{r,B,j}$ if and only if $Start(u, v) \leq End(u, v)$
3. Let $(u, v) \in CAND_{r,B,j}$. Then $G < PREV.dG$ and
4. Let $i \leq d$. Then $C_i \in CAND_{r,B,j}$ if and only if $G_{pred}(i) > E_{pred}(i)$, where

$$\begin{aligned} E_{pred}(i) &= C_i.dG + f_{Len}((2*C_i.Tmin - r) - (C_{i,y} - C_{i,x} + 2)); \\ G_{pred}(i) &= G + f_{Len}((2*C_i.Tmin - r) - (v - u + 2)); \end{aligned}$$

5. Let $i > d$. Then $C_i \in CAND_{r,B,j}$ if and only if $G_{succ}(i) > E_{succ}(i)$, where

$$\begin{aligned} E_{succ}(i) &= C_i.dG + f_{Len}((2*C_i.Tmax - r) - (C_{i,y} - C_{i,x} + 2)); \\ G_{succ}(i) &= G + f_{Len}((2*C_i.Tmax - r) - (v - u + 2)); \end{aligned}$$

6. Let $(u, v) \in CAND_{r,B,j}$ and $C_i \in CAND_{r,B,j-1}$ is the immediate predecessor of (u, v) in $CAND_{r,B,j}$. Then the value $Tmin$ corresponding to (u, v) can be found as

$$\begin{aligned}
Tmin &= \\
&= \min\{t \mid G + f_{Len}((2^*t - r) - (u - v + 2)) \leq C_i.dG + f_{Len}((2^*t - r) - (C_i.y - C_i.x + 2))\} = \\
&= Start(u, v; i)
\end{aligned}$$

7. Let $(u, v) \in CAND_{r,B,j}$ and $C_i \in CAND_{r,B,j-1}$ is the immediate successor of (u, v) in $CAND_{r,B,j}$. Then the value $Tmax$ corresponding to (u, v) can be found as

$$Tmax = \max\{t \mid G + f_{Len}((2^*t - r) - (u - v + 2)) < C_i.dG + f_{Len}((2^*t - r) - (C_i.y - C_i.x + 2))\}$$

Proof. We will consider in details only the Claim 1, 2. The other claims can be proved with similar technique based on Lemma 2.1, see Supplementary section 3.

1.

$$(u, v) \in CAND_{r,B,j} \tag{6.1}$$

Then $G \geq PREV.dG$ is impossible, otherwise, for all $(r - q, q)$ with $q \geq B+1$ we will have

$$\begin{aligned}
PREV.dG + f_{Len}((2^*q - r) - (PREV.y - PREV.x + 2)) &\leq \\
&< G + f_{Len}((2^*q - r) - (v - u + 2))
\end{aligned} \tag{6.2}$$

and therefore $(u, v) \notin CAND_{r,B,j}$.

2. (\rightarrow) Let's suppose that $(u, v) \in CAND_{r,B,j}$ and

$$Start(u, v) > End(u, v). \tag{6.3}$$

We shall show that (6.3) also contradicts (6.1). If (6.3) holds, then $Start(u, v) > B+1$.

Therefore $1 \leq d \leq N$ and thus

$$\begin{aligned}
Start(u, v) &= Start(u, v; d) = \\
&= \min\{C_d.Tmax + 1 \cup \dots \cup \{t \in [C_d.Tmin, C_d.Tmax] \\
&\quad \mid G + f_{Len}((2^*t - r) - (u - v + 2)) < C_d.dG + f_{Len}((2^*t - r) - (C_d.y - C_d.x + 2))\}\} \\
&\quad \}
\end{aligned}$$

For $d < N$ we have $C_{d+1}.Tmin - 1 = C_d.Tmax$ therefore, taking in account the definition of $End(u, v)$, (6.3) implies $d < N$;

$$\begin{aligned}
Start(u, v) &= C_d.Tmax + 1; \\
End(u, v) &= C_{d+1}.Tmin - 1 = C_d.Tmax = Start(u, v) - 1
\end{aligned} \tag{6.4}$$

or $d = N$;

$$\begin{aligned}
Start(u, v) &= C_d.Tmax + 1 = L + 1; \\
End(u, v) &= L
\end{aligned} \tag{6.5}$$

For brevity, we will consider only case (6.4). In this case $d < N$ and $C_{d+1}.Tmin = C_d.Tmax + 1$. According to (6.4), for all $t \in [C_d.Tmin, C_d.Tmax]$

$$G + f_{Len}((2^*t - r) - (u - v + 2)) \geq C_d.dG + f_{Len}((2^*t - r) - (C_d.y - C_d.x + 2)) \tag{6.6}$$

and for all $t \in [C_{d+1}.Tmin = C_d.Tmax+1, C_{d+1}.Tmax]$

$$G+f_{Len}((2^*t - r) - (u-v+2)) \geq C_{d+1}.dG+f_{Len}((2^*t - r)-(C_{d+1}.y-C_{d+1}.x+2)) \quad (6.7)$$

According to (6.1) let

$$(u, v) = OWNER_{r,B,j}(r-q, q) \quad (6.8)$$

for some $q \geq B+1$. We have to consider separately two cases: (i) $q \leq C_d.Tmax$ and (ii) $q \geq C_{d+1}.Tmin = C_d.Tmax+1$. In case (i). Lemma 2.1 (see Supplementary section 3) and (6.4) imply

$$G+f_{Len}((2^*q - r) - (u-v+2)) \geq C_d.dG+f_{Len}((2^*q - r)-(C_d.y-C_d.x+2)) \quad (6.9)$$

and in case (ii) Lemma 2.1 (see Supplementary section 3) and (6.5) imply

$$G+f_{Len}((2^*q - r) - (u-v+2)) \geq C_{d+1}.dG+f_{Len}((2^*q - r)-(C_{d+1}.y-C_{d+1}.x+2)) \quad (6.10)$$

Formulae (6.9), (6.10) contradict (6.8) and therefore contradict (6.1). This completes proof of the claim 2 of Lemma 3.2, case (->).

2 (<-) . Let

$$Start(u, v) \leq End(u, v). \quad (6.11)$$

Our goal is to prove that $(u, v) \in CAND_{r,B,j}$. Analogously to case 1a, (6.11) implies (cf. (6.4) and (6.5))

$$Start(u, v) = Start(u, v; d) = \min \{ t \in [C_d.Tmin, C_d.Tmax] \mid G+f_{Len}((2^*t - r) - (u-v+2)) < C_d.dG+f_{Len}((2^*t - r)-(C_d.y-C_d.x+2)) \} \quad (6.12)$$

$$End(u, v) = End(u, v; i) = \max \{ t \in [C_{d+1}.Tmin, C_{d+1}.Tmax] \mid G+f_{Len}((2^*t - r) - (u-v+2)) < C_{d+1}.dG+f_{Len}((2^*t - r)-(C_{d+1}.y-C_{d+1}.x+2)) \} \quad (6.13)$$

For example, let's (6.12) is true, case of (6.13) can be treated in the same way. Let $q = C_d.Tmax$. (6.13) implies

$$G+f_{Len}((2^*q - r) - (u-v+2)) < C_d.dG+f_{Len}((2^*q - r)-(C_d.y-C_d.x+2)) \quad (6.14)$$

But $(C_d.x, C_d.y) = OWNER_{r,B,j-1}(r-q, q)$ and therefore, for all $(x, y) \in D_{r,B,j-1}$

$$C_d.dG+f_{Len}((2^*q - r)-(C_d.y-C_d.x+2)) < C_i.dG+f_{Len}((2^*q - r)-(C_i.y-C_i.x+2)) \quad (6.15)$$

Formulae (6.14), (6.15) imply that $(u, v) = OWNER_{r,B,j-1}(r-q, q)$ and thus $(u, v) \in CAND_{r,B,j}$. This completes the proof of case 1b.

Section 7. The algorithm *MLF_E*.

Below we give the implementation of the Algorithm *MLF_E*. The presentation is extended compared to Supplementary, algorithm S1; the arrays *BACKWARDZONE*, *FORWARDZONE*, *VAR_MAINWORK*, *VAR_STRIPWORK* and the functions *InternalLoopMainRow(B)*; *InternalLoopStripRow_ES(B)* are given explicitly, see previous sections.

```
algorithm MLF_E( input: sequence RNA)
var global
  int BACKWARDZONE[L], FORWARDZONE[L];
  real VAR_MAINWORK[U], VAR_STRIPWORK[U];
begin
  // 1. Pre-processing
  1.1. for all base pairs  $(A, B) \in U$  do begin
    VAR_MAINWORK[A, B] :=  $\infty$ ;
    VAR_STRIPWORK[A, B] :=  $\infty$ ;
  end
  1.2. Compute BACKWARDZONE[B] and FORWARDZONE[B],  $B = 1, \dots, L$ 

  // The main loop
  2. for all B: ( $1 \leq B \leq L$ ) in increasing order do begin
    2.1 HairpinRow(B);
    2.2. SimpleLoopRow(B);
    2.3.1. InternalLoopMainRow(B);
    2.3.2. InternalLoopStripRow_ES(B);
    2.3.3. for all A:  $(A, B) \in ROW_B$  in decreasing order do begin
      VAR_MAINWORK[A, B] :=
        := min{ VAR_MAINWORK[A, B], VAR_STRIPWORK[A, B] }
    end
  2.4. OptimalLoopRow(B);

  end

  // Post-processing
  3. Find min{ VAR_MAINWORK[A, B] |  $(A, B) \in U$  }

end
```