Finding MFE Structures Formed by Nucleic Acid Strands in a Combinatorial Set

Mirela Andronescu and Anne Condon

The Department of Computer Science, U. British Columbia
androneslcondon@cs.ubc.ca

1 Introduction

When designing sets of DNA strands for biomolecular computations, it is often desirable to have a "structure free" combinatorial set of strands, that is, a set of long strands which do not form any secondary structure, obtained by concatenating short strands in the designed set.

The ability to computationally predict the combination in a combinatorial set of strands with lowest minumum free energy (MFE) secondary structure is also useful in design of strands for directed mutagenesis and SELEX experiments [3] – biochemical analyses of a library of nucleic acid sequences, to determine whether simple mutations of the sequences have desired properties. The input sequence sets can be represented as strings of characters (DNA or RNA nucleotides) and "wild cards", which can code for several different characters using IUPAC (International Union of Pure and Applied Chemistry) format or other format. In the case of SELEX, it is useful to be able to predict not only the combination whose minimum energy structure has lowest energy, but also other combinations with minimum energy structures of low value; this is the problem we address in this paper.

In earlier work, we described an algorithm, CombFold, that calculates which concatenated long strand in a combinatorial set forms the minimum free energy secondary structure with the lowest energy [2]. In this work, we extend that algorithm to output k secondary structures with the lowest minimum free energies, where k is specified by the user.

We use the following definitions and notation throughout.

- Let *word* denote an RNA or DNA sequence $w = v_1 v_2 \dots v_l$, where $v_i \in \{A, C, G, U\}$ for RNA and $v_i \in \{A, C, G, T\}$ for DNA. The orientation of the strand is from 5' to 3', unless otherwise stated. For example, `ACGCUAGGCA` is an RNA word of length 10.
- Let *set* denote a set of g *words* of the same length l. Formally we use the notation $S = \{w_1, w_2, \dots, w_g \mid length(w_i) = length(w_j), \forall i, j \in$

$\{1, \ldots, g\}, i \neq j\}$. The following *set* (displayed as a column of words) is formed of 4 words of length 5:

```
AUACG
UAGCG
GCCGA
CUGCG
```

The word order in a set does not matter, but for convenience later, we assume that the words in S are indexed and can be ranked by their index.

- Let *Input-Set* denote a sequence of s *sets*, $IS = S_1, S_2, \ldots, S_s$. For example, the following is an *Input-Set* of 5 sets:

```
UAGCGA    CAGCGUAAUAU    AUGCG    AUAGCGGUA    AUCG
AUAGAU    AGAUGCGCGGU             GAGCGCAAG    CUGC
          UAGGCUAGCGU                          GCGA
```

Note that the number of words in each *set* can differ, as can the length of the words across *sets*.
An *Input-Set* can also be written in terms of *words* rather than *sets*: $IS = \{w_{ij}, 1 \leq i \leq s, 1 \leq j \leq g_i, \forall i, 1 \leq i \leq s\}$, where g_i is the number of *words* in the *set* i.

$$\begin{array}{cccc} w_{11} & w_{21} & \ldots & w_{s1} \\ w_{12} & w_{22} & \ldots & w_{s2} \\ \vdots & \vdots & \vdots & \vdots \\ w_{1g_1} & & & \\ & \vdots & & w_{sg_s} \\ & w_{2g_2} & & \end{array}$$

Thus, an *Input-Set* IS is characterized by s sets, where each set S_i has g_i words, of length l_i. In what follows, when IS is fixed, we consider all its characteristics: s, w_{ij}, g_i, l_i, $\forall i, j, 1 \leq i \leq s, 1 \leq j \leq g_i$, to be known.

- Let *Combination* denote an RNA/DNA sequence, formed by concatenating one word w_{ij} of each set S_i from IS, starting at S_1 and finishing at S_s. For example $C = w_{11}w_{21}\ldots w_{s1}$ is a combination formed by concatenating the first word of each set together. Generally, a *combination* is of the form $C = w_{1b_1}w_{2b_2}\ldots w_{sb_s}$, where $1 \leq b_i \leq g_i$. Here, b_i denotes the word rank within the set S_i. A *combination* has the length $n = \sum_{i=1}^{s} l_i$. If we think of a *combination* as a sequence of nucleotides rather than a concatenation of words, we can denote it as $C = c_1c_2\ldots c_n$, with $c_i \in \{A, C, G, U\}$ for RNA.
- Given an *Input-Set* IS, the set of all possible *combinations* forms the *Combinatorial-Set*: $CS = \{w_{1b_1}w_{2b_2}\ldots w_{sb_s} \mid 1 \leq b_i \leq g_i\}$. Note that

all *combinations* have the same length: $n = \sum_{i=1}^{s} l_i$ and that CS has $g_1 \times g_2 \times \ldots \times g_s$ elements. If $g_i > 1, \forall i$, then the number of elements in CS is exponential in s.

The *optimal MFE combination problem* is: given an *RNA Input-Set* IS and a thermodynamic model M, predict which *combination*, out of all elements of the *Combinatorial-Set* CS formed from IS, folds to a pseudoknot-free secondary structure with the lowest minimum free energy.

An extension of the *optimal MFE combination problem* is to find the k best MFE combinations, rather than the optimal one only. The *k-suboptimal MFE combinations problem* is: given an *RNA Input-Set* IS and a thermodynamic model M, predict which k different *combinations*, out of all elements of the *Combinatorial-Set* CS formed from IS, fold to pseudoknot-free secondary structures with the lowest minimum free energies.

In this paper, we build on earlier work [2] to develop an algorithm for the k-suboptimal MFE combination problem. In section 2, we first review our dynamic programming algorithm which runs in polynomial time, for solving the *optimal MFE combination problem*. Then, in Section 3, we present our algorithm for the *k-suboptimal MFE combinations problem*. We provide a theoretical and empirical analysis of the optimal and k-suboptimal MFE combinations problems in Section 4 and show that both run in polynomial time.

2 Review of Algorithm for the Optimal MFE Combination Problem

Our *CombFold* algorithm [2] is based on the classical algorithm of Zuker and Stiegler [4] for finding the minimum free energy secondary structure of a single RNA strand.

One method to solve the optimal MFE combination problem is to create all possible *combinations* and then to run the Zuker-Stiegler algorithm on each of them. However, depending on the characteristics of the *Input-Set*, the number of *combinations* may be very big. If $g_i = g > 1, \forall i$, then there are g^s *combinations*. Since the Zuker-Stiegler algorithm runs in $\Theta(n^3)$ time, where n is the length of the combinations, this approach has running time complexity that is $\Theta(g^s n^3)$. More generally, the number of combinations is exponential in the number of sets which have at least two words. We have implemented this exhaustive search approach under the name of *ExhaustS*, which will be discussed in Section 4.

To avoid this exponential running time, we extended the Zuker-Stiegler algorithm. In the description that follows, we use indices i and j for the nucleotide positions (i.e. columns in Table 1) in a *combination* C. We use $s(i)$ and $s(j)$ to denote the sets in which c_i and c_j are positioned, respectively. We say that c_i and c_j belong to, or are in, the sets $s(i)$ and $s(j)$ respectively.

		123... ....i..... ....j.. ...n
b_j	1	UAGCGA CAGCGUAAUAU AUGCG AUAGCGGUA AUCG
b_i	2	AUAGAU AGAUGCGCGGU GAGCGCAAG CUGC
	3	UAGGCUAGCGU GCGA

Table 1. Example of a combinatorial set of short RNA sequences.

We use b_i and b_j to denote the indices (i.e. the rows in Table 1) of the words containing c_i and c_j within the sets $s(i)$ and $s(j)$. Given a set S, $g(S)$ returns the number of words in S. Hence, b_i can take $g(s(i))$ values. When the *Input-Set* IS and b_i are given, we let the base c_i at position i of a combination that is in column i and row b_i be given by the function $c_i = \mathit{Nucleotide}(IS, b_i, i)$. Table 1 shows an example of the nucleotides c_i and c_j.

Notation for substructure free energy values

We use the following notation to denote free energy values of various substructures; in our implementation, the values are stored in four-dimensional arrays.

- $W'(j)$ is the lowest minimum free energy of a structure formed from the first j nucleotides $c_1 c_2 \dots c_j$ of a *combination*. Consequently, $W'(n)$ contains the lowest minimum free energy of any structure formed by any combination in the *Combinatorial-Set* corresponding to the *Input-Set* IS.
- $W^c(b_j, j)$ is the lowest minimum free energy of a structure formed from the first j nucleotides of a *combination* in which b_j is the word index of the set $s(j)$.
- $V^c(b_i, b_j, i, j)$ is the lowest minimum free energy of a structure formed from a *combination* fragment $c_i \dots c_j$ starting at i and ending at j, and with fixed word indices b_i and b_j, assuming that $(c_i.c_j)$ is a base pair.
- $H^c(b_i, b_j, i, j)$ is the lowest free energy of a *combination* fragment $c_i \dots c_j$ in which b_i and b_j are fixed, assuming that $(c_i.c_j)$ closes a hairpin loop.
- $S^c(b_i, b_j, i, j)$ is the lowest free energy of a *combination* fragment $c_i \dots c_j$ in which b_i and b_j are fixed, assuming that $(c_i.c_j)$ closes a stacked loop.
- $VBI^c(b_i, b_j, i, j)$ is the lowest minimum free energy of the *combination* $c_i \dots c_j$ in which b_i and b_j are fixed, assuming that $(c_i.c_j)$ closes an internal loop.
- $VM^c(b_i, b_j, i, j)$ is the lowest minimum free energy of a *combination* fragment $c_i \dots c_j$ in which b_i and b_j are fixed, assuming that $(c_i.c_j)$ closes a multi-branched loop.
- $WM^c(b_i, b_j, i, j)$ is the lowest minimum free energy value of a *combination* fragment $c_i \dots c_j$ that forms part of a multi-branched loop, and is used to calculate VM^c values.

Recurrence relations

The array free energy values are calculated using several recurrence relations. In describing these here, we omit for clarity the calculations involving dangling ends and terminal AU penalties [1]. First, $W'(j)$ is the minimum of the values $W^c(b_j, j)$, over all possible b_j:

$$W'(j) = min_{b_j} W^c(b_j, j).$$

Here,

$$W^c(b_j, j) = \min \begin{cases} \min_{b_{j-1} \in X(\{b_j, j\}, \{j-1\})} W^c(b_{j-1}, j-1), \\ \min_{1 \le i < j; b_{i-1}, b_i \in X(\{b_j, j\}, \{i-1, i\})} \\ \quad (V^c(b_i, b_j, i, j) + W^c(b_{i-1}, i-1)) \end{cases}$$

where X is a function which returns the feasible range of words for all of the needed (unknown) indexes. For the first line, the word corresponding to $j-1$ depends on the sets to which j and $j-1$ belong, and on b_j:

$$X(\{b_j, j\}, \{j-1\}) = \begin{cases} \{b_j\} & , \quad \text{if } s(j-1) = s(j) \\ \{1, \ldots, g(s(j-1))\}, & \text{if } s(j-1) \neq s(j) \end{cases}$$

For the second line of the recurrence for $W^c(b_j, j)$, there are two word indices, b_{i-1} and b_i, that we have to find the ranges for:

$$X(\{b_j, j\}, \{i-1, i\}) = \\ \begin{cases} b_j, b_j & , \quad \text{if } s(i-1) = s(i) = s(j) \\ \{1, \ldots, g(s(i-1))\}, b_j & , \quad \text{if } s(i-1) \neq s(i) = s(j) \\ \{1, \ldots, g(s(i-1))\}, b_{i-1} & , \quad \text{if } s(i-1) = s(i) \neq s(j) \\ \{1, \ldots, g(s(i-1))\}, \{1, \ldots, g(s(i))\}, & \text{if } s(i-1) \neq s(i) \neq s(j) \end{cases}$$

In the first two lines of the equation for W^c above, the feasible values for b_{j-1} (first line), and b_{i-1}, b_i (second line), depend on one other index: j, and its corresponding b_j. However, in a more general case, there are p indexes with known b's, and q indexes with unknown b's, for which we want to find the feasible ranges. The number of *if* lines needed to specify the function X in the general case will be 2^{p+q-1}. Since we are using the nearest neighbour thermodynamic model, the highest values for p and q are $p = 4$ and $q = 4$ in the case of internal loops, and $p = 2$ and $q = 6$ in the case of multi-branched loops, yielding $2^7 = 128$ *if* lines. Instead of enumerating all of these lines in our code, we developed an algorithm to compute the ranges for unknown b's, for arbitrary values of p and q. This procedure is described next.

The function X calculates the ranges for the unknown b's, for any number of known and unknown indexes. Figure 1 gives the pseudocode for the X procedure. The input is comprised of two groups: the first group contains

Compute X Procedure

input: group of p indexes with known b's $\{b_{i_1} \dots b_{i_p}, i_1 \dots i_p\}$,
group of q indexes with unknown b's $\{j_1 \dots j_q\}$;
output: q groups $B_{j_1} \dots B_{j_q}$ corresponding to $\{j_1 \dots j_q\}$;

```
procedure Compute X
    order the indexes i's and j's;
    identify the sets S_1 ... S_m to which i's and j's belong;
    for (S = S_1 to S_m)
        if (there exists i_k in set S)
            foreach (j_u in set S)
                B_{j_u} = {b_{i_k}};
            end foreach;
        else
            j_v ← the smallest j in S;
            B_{j_v} = {1, ..., g(S)};
            foreach (j_u in set S, with j_u ≠ j_v)
                B_{j_u} = {b_{j_v}};
            end foreach;
        end if;
    end for;
    return B_{j_1}, ..., B_{j_q};
end procedure X.
```

Procedure 1: Pseudocode for the X procedure. Details are described in the text.

the known b's and the known indexes: $\{b_{i_1} \dots b_{i_p}, i_1 \dots i_p\}$. The input has the property that if $s(i_j) = s(i_{j+1})$ then $b_{i_j} = b_{i_{j+1}}$. The known b's help to determine the ranges of the unknown b's. The second group contains the indexes of the unknown b's, $\{j_1 \dots j_q\}$. First, we need to order all the values $i_1 \dots i_p, j_1 \dots j_q$. This is necessary for the second step, which identifies the sets corresponding to each index. The two extreme situations are: (1) all indexes are in the same set, and thus there will be only one possible configuration for the unknown b's; (2) all indexes are in different sets, hence there will be $g(s(j_1)) \times \dots \times g(s(j_q))$ possible values for the unknown b's.

Once we have identified the sets, for each set S, first we check whether there exists the index of a known b in this set. If so, then all the j's in S will have the corresponding, unknown, b's equal to the known b. No other option is available for these unknown b's, since the value for the known b is fixed. If no known b exists in S, then all the unknown b's in S will be in the range $\{1, \dots, g(S)\}$, with the constraint that they will have the same values, being in the same set. In other words, we can give a value to the b of the smallest index in S, and all the other b's in S will have the same value. The function X will return a group of values for the needed unknown b's.

An example of a particular situation, with the groups $\{b_i, b_j, i, j\}$ and $\{i+1, i+2, j-2, j-1\}$ as input, is presented in Table 2, where $s(i) \neq s(i+1) = s(i+2) \neq s(j-2) \neq s(j-1) = s(j)$. In this case, b_{i+1} will take values in the range $\{1, \ldots, g(s(i+1))\}$, b_{i+2} will take the value that b_{i+1} takes, b_{j-2} will be in the range $\{1, \ldots, g(s(j-2))\}$, and b_{j-1} equals b_j. Hence, for this particular situation, there will be $g(s(i+1)) \times g(s(j-2))$ terms over which to minimize.

i	$i+1$	$i+2$	$j-2$	$j-1$	j
$\mathbf{b_i}$	1	b_{i+1}	1	b_j	$\mathbf{b_j}$
	$\vdots$		$\vdots$		
	$g(s(i+1))$		$g(s(j-2))$		

Table 2. Example of choices for b values for a particular situation. The known b's are in bold. The vertical lines signify that the index to the left is in a different set from the index to the right.

Using function X to decide which are the possible values for each word, the remaining recurrence relations for *CombFold* are a logical extension of the corresponding recurrence relations for the Zuker-Stiegler algorithm. The recurrences use free energy values for hairpins, stacked pairs, and interior loops which we denote by $\Delta G\text{-}H^c(IS, b_i, b_j, i, j)$, $\Delta G\text{-}S^c(IS, b_i, b_j, b_{i+1}, b_{j-1}, i, j)$, and $\Delta G\text{-}I^c(IS, b_i, b_j, b_{i'}, b_{j'}, i, j, i', j')$, respectively.

The relations for V^c and H^c are straightforward:

$$V^c(b_i, b_j, i, j) = \begin{cases} +\infty & , \quad \text{for } i \geq j \\ \min(H^c(b_i, b_j, i, j), S^c(b_i, b_j, i, j), & \\ \quad VBI^c(b_i, b_j, i, j), VM^c(b_i, b_j, i, j)), & \text{for } i < j \end{cases}$$

$$H^c(b_i, b_j, i, j) = \Delta G\text{-}H^c(IS, b_i, b_j, i, j)$$

We omit the details of the calculation of hairpin free energies; the interested reader can find these in the M.Sc. thesis of Andronescu [1]. For the calculation of stacked loops, finding b_{i+1} and b_{j-1} is imposed again by the nearest neighbour model itself.

$$S^c(b_i, b_j, i, j) = \min_{b_{i+1}, b_{j-1} \in X(\{b_i, b_j, i, j\}, \{i+1, j-1\})} (\Delta G\text{-}S^c(IS, b_i, b_j, b_{i+1}, b_{j-1}, i, j) + V^c(b_{i+1}, b_{j-1}, i+1, j-1)).$$

The internal loop free energy calculation is a minimization over i' and j', i.e. the closing pair of the internal loop. Once i' and j' fixed, we calculate the free energy value for each possible $b_{i'}$ and $b_{j'}$:

$$VBI^c(b_i, b_j, i, j) = \min_{i<i'<j'<j} (\min_{b_{i'}, b_{j'} \in X(\{b_i, b_j, i, j\}, \{i', j'\})} (\Delta G\text{-}I^c(IS, b_i, b_j, b_{i'}, b_{j'}, i, j, i', j') + V^c(b_{i'}, b_{j'}, i', j')))$$

The free energy for multi-loops adds the minimization over the necessary b's as well. The equations for WM^c and VM^c follow, where $\mathcal{M}_a$, $\mathcal{M}_b$, and $\mathcal{M}_c$ are penalties for multi-loops, branches, and unpaired bases that determine the standard multi-loop energy function. For $i < j$,

$$WM^c(b_i, b_j, i, j) = \min \begin{cases} V^c(b_i, b_j, i, j) + \mathcal{M}_b, \\ \min_{b_{i+1} \in X(\{b_i, b_j, i, j\}, \{i+1\})}(WM^c(b_{i+1}, b_j, i+1, j) + \mathcal{M}_c), \\ \min_{b_{j-1} \in X(\{b_i, b_j, i, j\}, \{j-1\})}(WM^c(b_i, b_{j-1}, i, j-1) + \mathcal{M}_c), \\ \min_{i \le h < j; b_h, b_{h+1} \in X(\{b_i, b_j, i, j\}, \{h, h+1\})}(WM^c(b_i, b_h, i, h) + \\ \quad WM^c(b_{h+1}, b_j, h+1, j)) \end{cases}$$

$$VM^c(b_i, b_j, i, j) = \mathcal{M}_a + \min_{i<h<j-1; b_{i+1}, b_h, b_{h+1}, b_{j-1} \in X(\{b_i, b_j, i, j\}, \{i+1, h, h+1, j-1\})} (WM^c(b_{i+1}, b_h, i+1, h) + WM^c(b_{h+1}, b_{j-1}, h+1, j-1))$$

In the implementation of our software *CombFold v1.0*, we did not implement the equation for VM^c as described above. This equation contains the sum of two WM^c terms in order to make sure that the multi-loop obtained has at least three branches (including the closing one), at the cost of increased complexity, i.e. n^3 instead of n^2 for computing VM^c (see also section 4). In our implementation, $VM^c = \mathcal{M}_a + WM^c(b_{i+1}, b_{j-1}, i+1, j-1)$ where $b_{i+1}, b_{j-1} \in X(\{b_i, b_j, i, j\}, \{i+1, j-1\})$, while we used a mechanism to make sure that the predicted multi-loops have at least three branches. We believe that this does not involve significantly different predictions, and we plan to implement the more accurate formula above in the next version of *CombFold*.

3 An algorithm for the *k*-suboptimal MFE combinations problem

The algorithm for the *optimal MFE combination problem*, just described in the previous section, returns only the combination which has the smallest MFE. We next describe how the algorithm can be extended to return the k combinations that have the lowest MFE.

Suppose that the *Input-Set* IS contains s sets S_i, each having g_i words. We will add the superscript "(1)" to the notation of our sets to denote that first

we are looking for the optimal combinations. The superscripts for the next combinations will be "(2)" and so on. Thus, $IS^{(1)} = \{S_1^{(1)}, S_2^{(1)}, \ldots, S_s^{(1)}\}$ will be associated with the *Combinatorial-Set* $CS^{(1)}$.

First, we find the optimal MFE combination using the method described in the previous section. Let the combination $C^{(1)} = w_{1C_1}^{(1)} w_{2C_2}^{(1)} \ldots w_{sC_s}^{(1)}$ denote the optimal MFE combination, where C_i denotes the index of the word in the set $S_i^{(1)}$, which belongs to the optimal combination. The *Input-Set* $IS^{(1)}$ contains all the possible combinations of the original set IS. To find the next best combinations, first we partition the set $IS^{(1)}$ into s sets which do not contain $C^{(1)}$:

$$
\begin{aligned}
IS^{(2)1} &= \{\ S_1^{(1)} - \{w_{1C_1}^{(1)}\}, \quad S_2^{(1)}, \quad \ldots \quad , S_s^{(1)} \quad \} \\
IS^{(2)2} &= \{\quad \{w_{1C_1}^{(1)}\}, \quad S_2^{(1)} - \{w_{2C_2}^{(1)}\}, \ldots \quad , S_s^{(1)} \quad \} \\
&\vdots \\
IS^{(2)s} &= \{\quad \{w_{1C_1}^{(1)}\}, \quad \{w_{2C_2}^{(1)}\}, \quad \ldots\ , S_s^{(1)} - \{w_{sC_s}^{(1)}\}\ \}
\end{aligned}
$$

For convenience later, we denote the newly created sets with $S_i^{(2)j}$, where $1 \leq i, j \leq s$, i denotes the set index within the *Input-Set*, as in the previous notations, and j denotes the index of the newly created *Input-Set*:

$$
\begin{aligned}
IS^{(2)1} &= \{S_1^{(2)1}, S_2^{(2)1}, \ldots, S_s^{(2)1}\} \\
IS^{(2)2} &= \{S_1^{(2)2}, S_2^{(2)2}, \ldots, S_s^{(2)2}\} \\
&\vdots \\
IS^{(2)s} &= \{S_1^{(2)s}, S_2^{(2)s}, \ldots, S_s^{(2)s}\}
\end{aligned}
$$

The *Input-Sets* $IS^{(2)1}, IS^{(2)2}, \ldots, IS^{(2)s}$ have the following properties:

- $C^{(1)} \notin CS^{(2)m}, \forall m, 1 \leq m \leq s$;
- $CS^{(2)m} \cap CS^{(2)m'} = \emptyset, \forall m, m', 1 \leq m, m' \leq s, m \neq m'$;
- $\{C^{(1)}\} \cup CS^{(2)1} \cup \ldots \cup CS^{(2)s} = CS^{(1)}$,

where $CS^{(i)j}$ denotes the *Combinatorial-Set* associated with the *Input-Set* $IS^{(i)j}$. In other words, (1) the combination $C^{(1)}$ is not included in any of the new *Input-Sets* created by the partitioning process, (2) the new input sets do not have any combinations in common and (3) the whole space of combinations in $CS^{(1)}$ is covered by the new input sets plus the optimal combination found. This leads to finding the optimal combinations for each of $IS^{(2)1}, IS^{(2)2}, \ldots, IS^{(2)s}$, followed by choosing the one with the smallest MFE. Thus, the free energy of the second best combination, i.e. the combination with the second lowest MFE, will be $\Delta G^{(2)} = \min(\Delta G^{(2)1}, \Delta G^{(2)2}, \ldots \Delta G^{(2)s})$,

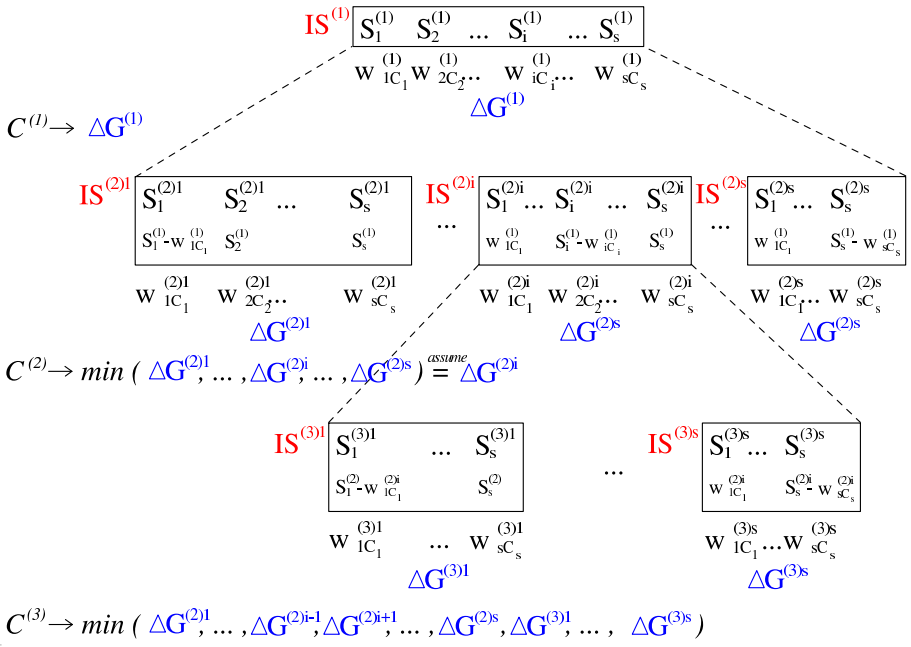

Fig. 1. The algorithm for finding the k suboptimal MFE combinations of a combinatorial set.

where $\Delta G^{(2)i}$ is the MFE of the optimal combination of $IS^{(2)i}$. Let i be such that $\Delta G^{(2)} = \Delta G^{(2)i}$ and let $C^{(2)} = w^{(2)i}_{1C_1} w^{(2)i}_{2C_2} \dots w^{(2)i}_{sC_s}$ denote the second best combination. The next step is to partition $IS^{(2)i}$, in the same way we partitioned $IS^{(1)}$. We will obtain the *Input-Sets* $IS^{(3)1}, IS^{(3)2}, \dots IS^{(3)s}$. Now, note that the following are true:

- $C^{(1)}$ and $C^{(2)} \notin CS^{(2)m}$ and $CS^{(3)m'}, \forall m, m', 1 \leq m, m' \leq s, m \neq i$;
- $CS^{(a)m} \cap CS^{(b)m'} = \emptyset$, for $a, b \in \{2, 3\}$ and $m, m' \in \{1, \dots, s\}$, with either $a \neq b$ or $m \neq m'$ (or both);
- $\{C^{(1)}, C^{(2)}\} \cup CS^{(2)1} \cup \dots \cup CS^{(2)i-1} \cup CS^{(2)i+1} \cup \dots \cup CS^{(2)s} \cup CS^{(3)1} \cup \dots \cup CS^{(3)s} = CS^{(1)}$.

Thus, $\Delta G^{(3)}$, the MFE of the third combination, will be

$$\min(\Delta G^{(2)1}, \dots, \Delta G^{(2)i-1}, \Delta G^{(2)i+1}, \dots, \Delta G^{(2)s}, \Delta G^{(3)1}, \dots, \Delta G^{(3)s}).$$

Figure 1 shows the steps just described. Recursively continuing in the same way, we can find the best k combinations. However, note that the tree of partitioned *Input-Sets* will grow proportionally with k, more exactly, it will have a number of leaves that is at most ks, which implies increase in run time and space.

It is important to note that when creating the new *Input-Set*s $IS^{(i)j}$, $1 \leq j \leq s$ the sets with a lower index j will typically have a bigger solution space (i.e. number of possible combinations) than the ones with a higher index. Thus, if after we found the second combination, $\Delta G^{(2)}$ equals $\Delta G^{(2)s}$, the third combination will be found much more quickly than if $\Delta G^{(2)}$ equals $\Delta G^{(2)1}$. Also, it is possible that the *Input-Set* which has the next best combinations will be partitioned in less than s partitions (or even no partitions at all), since the other partitions are empty. In this case, only the optimal MFE combinations of the non-empty partitions will be considered. Examples of the running time on some problem instances are discussed in Section 4.

4 Time and space complexity

Extending the $O(n^3)$ algorithm for secondary structure prediction of single nucleic acid molecules, the *optimal MFE combination* algorithm traverses the *Input-Set* in the same way, but for each position i and j, several possibilities might exist. We consider that the number of words g_i in each set S_i is limited by a constant bound g_{max}, and we measure the complexity in terms of the combinations length: $n = l_1 + l_2 + \ldots + l_s$. Also, we consider that the ranges returned by the X function is bounded by a constant and will be omitted from the theoretical analysis. In practice, the number of words in each set, the number of sets, the length of the words in each set, as well as the nucleotides composing the set, all have an impact on the run time. First we give an analysis of the theoretical complexity, and later in this section we will analyse the *CombFold* implementation on several specific *Input-Sets*.

Theoretical analysis

The theoretical time complexity of calculating each array described in Section 2 in the worst case follows:

- W': $O(g_{max}n)$, because for each j calculated in W^c, we minimize over all possible words of j, and there are at most g_{max} such words;
- W^c: $O(g_{max}^5 n^2)$, because for each $j, 1 \leq j \leq n$ there are at most g_{max} possibilities, and we minimize over i. When dangling ends are included, i and j's neighbours may have unknown b's, leading to four options for unknown b's (details omitted). However, b_{i-1}, b_i and b_{i+1} can only be in different words if the length of the word $l(s(i))$ is 1. But if $l(s(i)) = 1$, $g(s(i))$ is at most 4 (because there are 4 different nucleotides), no matter what the value of g_{max} is;
- V^c: $O(g_{max}^2 n^2)$, because for each i and j, we minimize over a constant number of terms, and for each i and j there are at most g_{max} possibilities;
- S^c: $O(g_{max}^4 n^2)$, because for each i, j and their corresponding b_i and b_j, we minimize over potential different values for b_{i+1} and b_{j-1};

- H^c: $O(g_{max}^4 n^2)$, because for each i, j and their corresponding b_i and b_j, the term which has the greatest complexity has minimization over 4 terms, but 2 of them happen only if the word length is 1, so they are reduced to constant times;
- VBI^c: $O(g_{max}^8 n^4)$, but we assume the internal loops do not have more than a constant number of bases (e.g. 30) on each side between the branches, and thus the complexity for internal loops becomes $O(g_{max}^8 n^2)$. The power of 8 comes from the most general case of internal loops;
- WM^c: $O(g_{max}^4 n^3)$, because the most costly branch of the WM^c calculation for each i and j is to find the best h for multi-loop partitioning. Each of i, j and h are in at most g_{max} words;
- VM^c: $O(g_{max}^8 n^3)$, because for each i and j we minimize over h, and when we include all dangling ends, there are two known b's and six unknown b's in the worst case.

Thus, if we consider both g_{max} and n in our analysis, the worst case time complexity is $O(g_{max}^8 n^3)$. In practice, g_{max} is often considered a constant, which leads to complexity proportional to n^3. The arrays W', W^c, V^c and WM^c need to be stored in memory. The space complexity is $O(g_{max}^2 n^2)$, or $O(n^2)$ if we consider g_{max} a constant.

The worst theoretical time complexity of the k-suboptimal MFE combinations problem is $O(skg_{max}^8 n^3)$ and the worst space complexity is $O(skg_{max}^2 n^2)$. However, in practice, some of the *Input-Sets* after partitioning become empty.

Empirical analysis

We compared the running time performance of *CombFold v1.0* with suboptimal predictions with that of *ExhaustS*, a simple (exponential time) exhaustive search algorithm, which creates all possible combinations and for each, calculates its minimum free energy using *SimFold* [1], our implementation of the Zuker-Stiegler algorithm. For *Input-Sets* with a small number of combinations, it is expected that *CombFold* takes more time and space than *ExhaustS*, because *CombFold* is a more complex algorithm. However, although the space is not a problem for *ExhaustS*, the running time quickly grows and becomes impractical.

Figure 2 gives the run time performance of *CombFold* with $k = 1, 2, 3, 10$ and *ExhaustS* on randomly generated *Input-Sets* of different characteristics. All the tests have been performed on machines with CPU Pentium III 733 MHz, memory cache 256 KB and RAM memory 1GB, running Linux 2.4.20. All graphs show the CPU time in seconds, presented on a log scale, versus variation of different characteristics of the *Input-Sets*. To simplify the analysis, we chose $g_1 = \ldots = g_s = g$ and $l_1 = \ldots = l_s = l$, and we took variations of s, g and l. Having all set sizes equal and all set lengths equal, the number of combinations will be g^s, and the length of the combinations will be $l \cdot s$.

The graph in (a) shows a comparison between the running time of *CombFold* with $k = 1, 2, 3, 10$ and *ExhaustS*, on a set of 19 instances having g and

l fixed at 2 and 10, respectively. The number of sets s varies from 1 to 19, yielding $2^1 = 2$ combinations of length 10 to $2^{19} \approx 0.5 \cdot 10^6$ combinations of length 190. *CombFold* with $k = 1$ becomes faster than *ExhaustS* at $s = 8$, with $k = 2$ and 3 becomes faster at $s = 10$, and *CombFold* with $k = 10$ becomes faster at $s = 12$. Note that the slope of the curves suggest that *CombFold* grows polynomially, while *ExhaustS* grows exponentially in s.

The graph in (b) shows a similar situation as in graph (a), but when g is fixed at 3 rather than 2, $l = 10$ and s takes values in the range 1 to 12, leading to $3^1 = 3$ combinations of length 10 to $3^{12} \approx 0.5 \cdot 10^6$ combinations of length 120. The number of combinations being bigger for the same s, *CombFold* with $k = 1$ outperforms *ExhaustS* when $s = 6$, with $k = 2$ and 3 when $s = 7$, and with $k = 10$ when $s = 8$.

Graph (c) shows a comparison when s and l are fixed to 6 and 10 respectively, but g varies from 1 to 13. These yield $1^6 = 1$ to $13^6 \approx 4.8 \cdot 10^6$ combinations of length 60. Note that in this case *ExhaustS* grows polynomially in g, however, it grows more quickly than *CombFold*. Indeed, the graph shows that *CombFold* with $k = 1$ becomes faster than the *ExhaustS* when $g = 3$, with $k = 2$ and 3 when $g = 4$ and with $k = 10$ when $g = 5$.

Graph (d) gives the comparison when s and g are fixed to 8 and 2, respectively, leading to a fixed number of $2^8 = 256$ combinations. However, the length of the words vary from 10 to 100, yielding combinations of length 80 to 800. Again, *ExhaustS* grows more quickly, but still polynomially, only the length of the combinations being changed. *ExhaustS* is outperformed by *CombFold*($k = 1$) at $l = 10$ and by *CombFold*($k = 2$) at $l = 50$. On the instances we tested, *ExhaustS* outperforms *CombFold* with $k = 10$, and becomes roughly the same speed as *CombFold* with $k = 3$ when $l = 100$.

On all these four graphs, we note that *CombFold* with $k = 1$ and 2, and *ExhaustS* are nicely curved, while *CombFold* with $k = 3$ and 10 have "hills" and "valleys". To see how the curves look like, we created two sets of 50 instances of *Input-Sets* with exactly the same characteristics: graph (e) with $s = 10, g = 3, l = 5$ and graph (f) with $s = 8, g = 8, l = 4$. The results comfirm the explanation we gave earlier in Section 3: When $k = 1$, *CombFold* fills all the arrays, a small variation happening due to the distribution of the nucleotides in the words. When $k = 2$, the arrays for s more sets are always calculated, no matter what the optimal combination is. However, depending on which the second best combination is, the size of the next *Input-Sets* that partition the solution space can differ substantially. This influence propagates on to the next best combinations, such that when $k = 10$, the differences in time between different instances can vary substantially. Also, note that for some instances, the time for $k = 3$, and even for $k = 10$, is very close or equal to the time for $k = 2$. This means that the second best combination was part of a very small *Input-Set*, which was partitioned in fewer (or even 0) non-empy *Input-Sets*. The graphs also show the run time of the exponential algorithm. For graph (e) there are $3^{10} \approx 60,000$ combinations of length 50, and *ExhaustS* is more than one order of magnitude slower than *CombFold* with

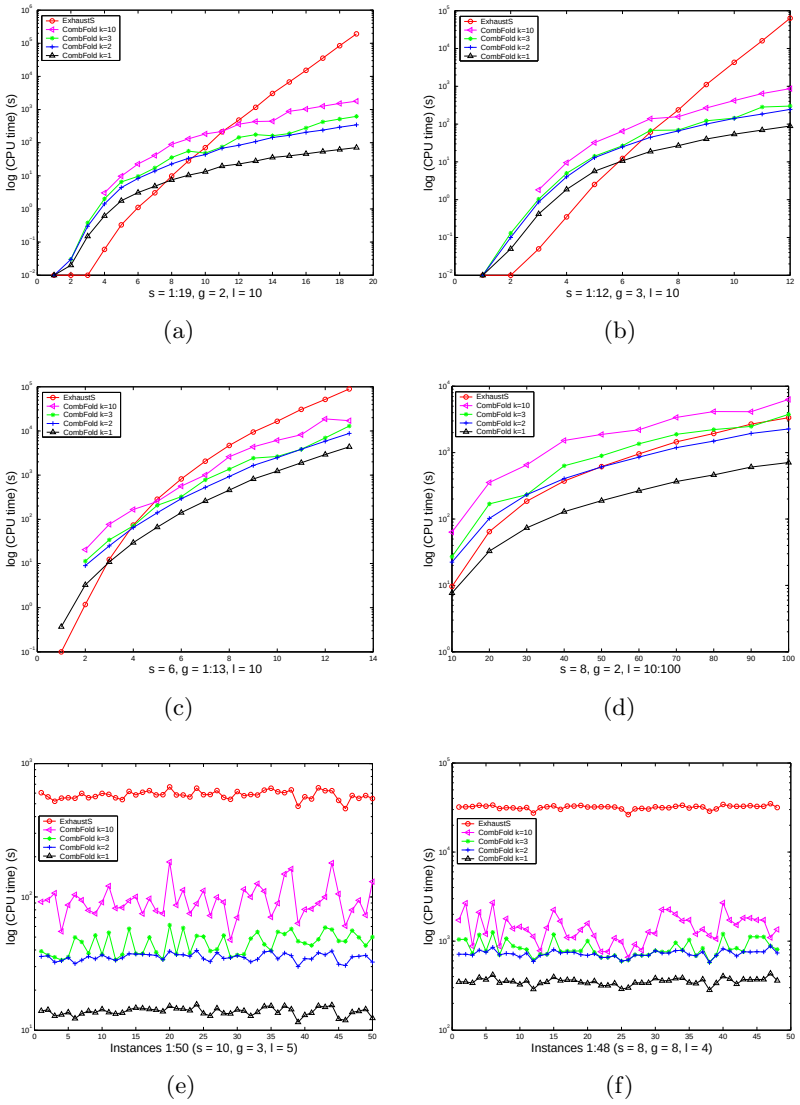

Fig. 2. Performance of *CombFold* with $k = 1, 2, 3, 10$ and *ExhaustS*, on sets with different characteristics: (a) 19 instances with s ranging from 1 to 19, and the same $g = 2$ and $l = 10$; (b) 12 instances with s ranging from 1 to 12, and the same $g = 3$ and $l = 10$; (c) 13 instances with g ranging from 1 to 13, and the same $s = 6$ and $l = 10$; (d) 10 instances with l ranging from 10 to 100, and the same $s = 8$ and $g = 2$; (e) 50 instances with the same characteristics: $s = 10, g = 3, l = 5$; (f) 48 instances with the same characteristics: $s = 8, g = 8, l = 4$.

$k = 1$, and 5-6 times slower than *CombFold* with $k = 10$. For graph (f), where the number of combinations is $8^8 \approx 16.8 \cdot 10^6$ of length 32, the exponential algorithm is substantially slower, being about two orders of magnitude slower than *CombFold*$(k = 1)$, and more than one order of magnitude slower than *CombFold*$(k = 10)$.

5 Conclusions

We presented here an algorithm that, given a combinatorial set and parameter k, predicts the k secondary structures with lowest minimum free energies in the combinatorial set. When the number of words in each set of the overall input-set is considered to be a constant, our algorithm runs in $O(skn^3)$ time.

In our algorithms, given a combination C, we look at the minimum free energy structure only. Extensions of these problems would be to find suboptimal structures (i.e. whose free energy is greater than the MFE), or to consider pseudoknots. Another problem for future work would be to find an algorithm with better running time, for example $O(n^3 + k)$.

References

1. M. Andronescu, *Algorithms for predicting the Secondary Structure of pairs and combinatorial sets of nucleic acid strands*, M.Sc. Thesis, U. British Columbia, November 2004. http://www.cs.ubc.ca/grads/resources/thesis/Nov03/Mirela_Andronescu.pdf
2. M. Andronescu, D. Dees, L. Slaybaugh, Y. Zhao, B. Cohen, A. Condon, and S. Skiena, *Algorithms for testing that sets of DNA words concatenate without secondary structure*, Lecture Notes in Computer Science, 2568 Springer 2003, 182-195. Revised version appeared in Natural Computing, 2(4):391–415, 2003.
3. J. V. Ponomarenko, G. V. Orlova, A. S. Frolov, M. S. Gelfand and M. P. Ponomarenko, *SELEX_DB: a database on in vitro selected oligomers adapted for recognizing natural sites and for analyzing both SNPs and site-directed mutagenesis data*, Nucl. Acids. Res. (2002); 30 (1): 195-199.
4. M. Zuker and P. Stiegler, *Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information*, Nucl. Acids. Res. (1981) 9: 133-148.