GRAPH TREE DECOMPOSITION ENABLED BIOPOLYMER FOLDING

by

QI LI

(Under the Direction of Liming Cai)

ABSTRACT

Biopolymer tertiary structure prediction by computer programs plays a very important role in complementing the experimental determination method. There are two structure prediction approaches: template-based and *ab initio* predictions. Due to the nature of residue interactions in biopolymer tertiary structures, both prediction approaches are required to perform intensive computations. Previous research has discovered a small treewidth property for interaction topology graphs of biopolymer tertiary structures, rendering the opportunity to speed up the combinatorial computation needed by the predictions with graph tree decomposition based dynamic programming. In the current research, a heuristic strategy is developed to reduce the memory space usage for the dynamic programming. An application of this method to the template-based protein tertiary structure prediction is considered in detail. In addition, the method is extended as a step toward the *ab initio* prediction of biopolymer tertiary structures.

INDEX WORDS: template-based structure prediction, *ab initio* structure prediction, biopolymer sequence-structure prediction, graph tree decomposition, treewidth, dynamic programming

# GRAPH TREE DECOMPOSITION ENABLED BIOPOLYMER FOLDING

by

QI LI

B.S., Central China Normal University, China, 2001

M.S., East China Normal University, China, 2004

M.Ed., University of Georgia, 2006

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2010

# GRAPH TREE DECOMPOSITION ENABLED BIOPOLYMER FOLDING

by

QI LI

| | |
|---|---|
| Major Professor: | Liming Cai |
| Committee: | Russell Malmberg<br>Khaled Rasheed |

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
December 2010

# ACKNOWLEDGEMENTS

It has been my honor to study under Dr. Liming Cai's guidance. Dr. Cai is the most dedicated professor I have been working with. I thank Dr. Cai for his patience, encouragement, and clear direction in this research, and his prompt response to requests.

I would like to thank Dr. Russell Malmberg and Dr. Khaled Rasheed for their valuable time and suggestions.

I would like to thank Xingran Xue, Joseph Robertson, Yingfeng Wang, Zhibin Huang and other members in the RNA Informatics group for their help.

Finally, I would like to thank my beautiful wife, Jin Tang, and both our parents, brothers and sisters for their endless support and love that make everything possible.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

# CHAPTER 1

# INTRODUCTION

Biopolymer (e.g. a protein or RNA) tertiary structure carries the essential information for defining the biopolymer biological functions. The tertiary structure determination of a biopolymer can be conducted by the use of X-ray crystallography and nuclear magnetic resonance (NMR) [1]. While experimental determination of a biopolymer tertiary structure and function is the most reliable way [10], a much faster, perhaps less accurate, structure prediction by computer programs can facilitate the experimental determination by narrowing down the number of structure candidates for further validation. This has been a strong motivation to develop accurate and efficient computational methods for biopolymer tertiary structure prediction from sequence data.

In general, the existing computational methods for biopolymer tertiary structure prediction can be either template-based or *ab initio*. A template-based prediction method uses templates as references to predict the structure of a new sequence through sequence-structure alignment [8, 9, 16, 33, 34, 35, 36]. A tertiary structure template is usually constructed from a consensus structure of homologous polymers. The template-based prediction assumes that the total number of different structural folds in nature is relative small compared with the number of sequences of biopolymers [35]. With known tertiary structures of biopolymers identified, the tertiary structure of a new sequence may have existed among these previously discovered structures. However, this method can only predict structures that we already knew. As an

alternative, a *ab initio* prediction method derives the tertiary structure directly from the sequence without referring to any previously solved structures [2, 10, 12, 13, 15, 19, 20, 21, 24, 26]. The *ab initio* prediction searches a space of 3D conformations of a new sequence for the most possible conformation with biological constraints. For example, the constraint for a protein sequence is that the native structure of the sequence should possess the minimum global free energy [35]. However, due to the large space 3D conformations, identifying the desired conformation is computation intensive. The *ab initio* prediction tends to trade efficiency with prediction accuracy.

The need for computation efficiency also exists in a template-based prediction since an efficient sequence-structure alignment algorithm is required to search a large number of structure templates. We are interested in examining how a graph tree decomposition framework can be used for the alignment algorithm. The graph tree decomposition can be very promising in terms of computational time. The tree decomposition has the beauty of separating unrelated concerns, which decreases the computational complexity from exponential time to polynomial time on graphs with a fixed treewidth. This can be accomplished through a bottom up dynamic programming, which carries out the optimal search process considering the biological constraints based on the tree bags of the tree decomposition of the examined graph. We call such a process tree decomposition based dynamic programming, or TDDP in short.

The TDDP has been studied for a template-based prediction for protein threading and RNA secondary structure homology search [14, 18, 28, 29, 32, 34, 35]. The core computation of these applications is the sequence-structure alignment using TDDP [29]. However, for a naïve implementation, the memory space for saving the dynamic tables can grow exponentially as the treewidth. In this research, we will propose a heuristic strategy to reduce the memory space of

the dynamic tables without sacrificing the running time complexity. In addition, we will explore the energy functions which will be used to compute the objective function during the dynamic programming process.

The TDDP can also be suitable for a *ab initio* structure prediction. The new challenge is that the dynamic programming process for the *ab initio* prediction has to consider not only the fitness score but also the geometric constraints due to the lack of folded templates. We will explore ideas on how to take advantage of the tree decomposition to integrate the geometric constraints into the dynamic programming process to keep the computation complexity as polynomial time. In this research, we will focus on an application to RNA 3D folding.

The rest of the thesis is organized as follows. Chapter 2 introduces the notion of graph tree decomposition and dynamic programming algorithm using a maximum independent set problem as an example. Chapter 3 explores the components of biopolymer folding in detail and how a TDDP algorithm can solve the alignment problem in a polynomial time for a given fixed treewidth. Chapter 4 illustrates the heuristic strategy to reduce the size of dynamic tables and the energy functions which are involved for template-based protein tertiary structure prediction. Chapter 5 examines a potential framework to integrate the check of the geometric constraints into a TDDP algorithm for the *ab initio* RNA 3D Folding. Chapter 6 concludes the work of this research and addresses the future work.

# CHAPER 2

# TREE DECOMPOSITION BASED DYNAMIC PROGRAMMING

In graph theory, a tree decomposition is a topological view of a graph with a tree representation. The tree decomposition gives rise to the notion of treewidth, a metric measuring how much the graph is tree-like. In algorithmic graph theory, the tree decomposition has been used to improve the efficiency of algorithms solving many combinatorial optimization problems on graphs constrained by a small treewidth.

The treewidth is usually the maximum number of vertices minus one in the tree nodes for a tree decomposition. While it is NP-complete to determine whether the treewidth of a given graph is at most a given integer $k$ [3, 4], there are polynomial approximation time algorithms for the problem on a bounded constant $k$ [4, 5, 6, 7]. The major interest in the tree decomposition is that we can solve many computationally hard problems for an arbitrary graph in polynomial or linear time if we have a tree decomposition of this graph with a treedwidth bounded by a fixed constant [4].

In this chapter, we will review the definition of the tree decomposition and related properties. Then we will illustrate how the tree decomposition speeds up solving the maximum independent set problem by a bottom up dynamic programming, which is also called TDDP as we did in chapter 1.

## 2. 1 Definition and Properties

Definition [25]: Let G = (V, E) be a graph, where V is the vertex set and E is the edge set. A tree decomposition of a graph G is a pair (X, T), where X = $\{X_i \mid i \in I\}$ is a family of subsets of V, and T is a tree whose nodes are annotated with integers in I, satisfying the following conditions:

1) $\bigcup_{i \in I} X_i = V$
2) $\forall\, u, v, (u, v) \in$ E, $\exists$ i $\in$ I such that $u \in X_i$ and $v \in X_i$
3) $\forall\, i, j, k \in$ I, if node $j$ is on the path from node $i$ to node $k$ in T, then $X_i \cap X_k \subseteq X_j$

The tree node $i$ of T is associated with $X_i$, which is called a tree bag. The first condition requires that each graph vertex should be associated with at least one tree node. The second condition requires that two adjacent vertices in the graph should appear in at least one same tree node. The third condition requires that the tree nodes containing a vertex form a connected subtree of T.

There could be many tree decompositions for a graph. For example, a trivial tree decomposition contains all vertices of a graph in a single root node. The treewidth of the tree decomposition (X, T) is defined as the $max_{i \in I}|X_i|$-1. The optimal tree decomposition of a graph G is the minimum treewidth over all possible tree decompositions. The optimal tree decomposition can have many forms as well. Figure 2.1 gives an example about a graph and its tree decomposition of width 2.

A very important characteristic of the tree decomposition is that the tree bag on the path from one tree bag to another tree bag is a graph separation. In Figure 2.1(b), the tree bag 3 directly connected the tree bag 1, the tree bag 2 and the tree bag 4 together. If we remove the common vertices {b, d, e} contained in the tree bag 3, the remaining vertices {a, c, g} are separated. This characteristic with other properties make it possible to develop dynamic programming algorithms for many graph problems with a tree decomposition. In the following, we give a

detailed explanation about the dynamic programming process on how the tree decomposition solves the maximum independent set problem for a graph given its tree decomposition.

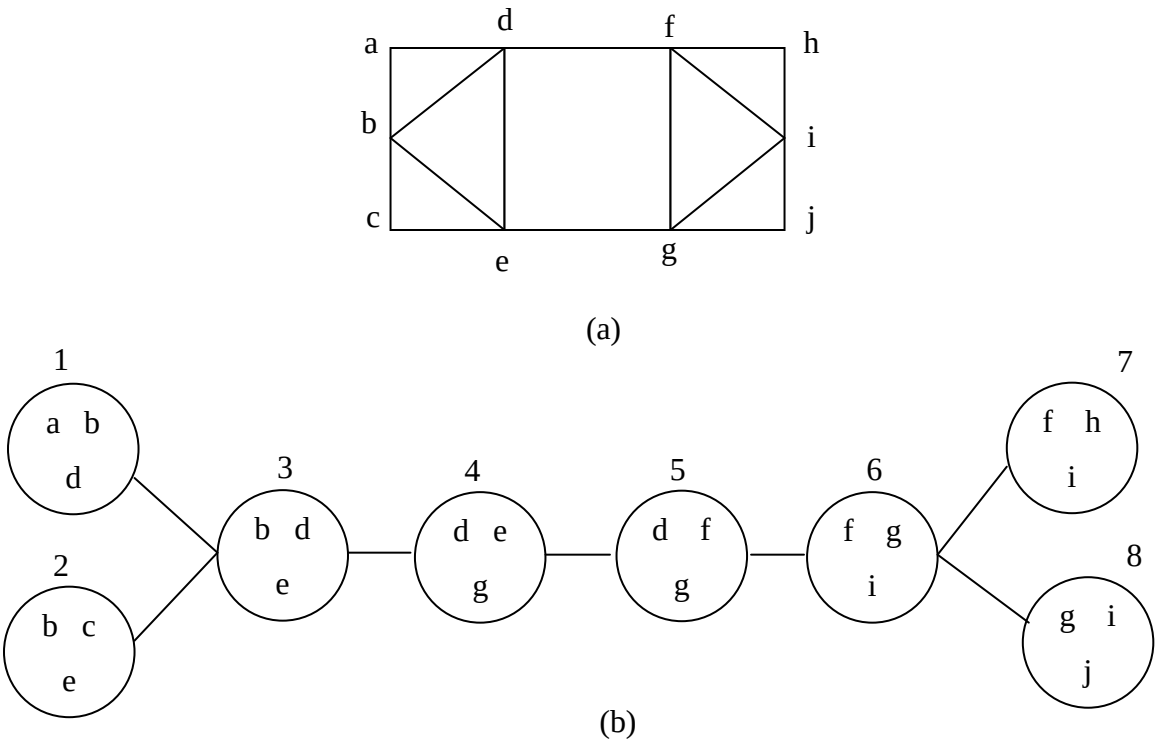


Figure 2.1: A graph (a) and its tree decomposition (b) of width 2.

**2. 2 A TDDP Algorithm**

The maximum independent set problem finds the maximum size of a set W ⊆ V in a given graph G = (V, E), such that for all $u, v \in$ W and $(u, v) \notin$ E. Assume we have a tree decomposition ($\{X_i \mid i \in$ I}, T) of the input graph G with the treewidth $k$. For each $i \in$ I, define $Y_i = \{v \in X_j \mid$ $j = i$ or $j$ is a descendant of $i\}$. Based on the separation property and the definition of the tree decomposition, we argue that, if we have an independent set W of the subgraph G[$Y_i$] induced by $Y_i$, we only need consider what vertices in $X_i$ belong to W not what vertices in $Y_i - X_i$ when we

extend W to an independent set of G. This builds the foundation to solve the maximum independent set problem using a bottom up dynamic programming algorithm in O($m2^k$) running time, where $m$ is the number of tree nodes.

The dynamic programming starts with building one table $T_i$ for each tree bag $X_i$. The table $T_i$ contains $|X_i|$ + 3 columns. One column for each vertex in $X_i$ . The extra three columns are "Valid", "Number" and "Optimal". The "Valid" column checks if a specific set in a row is valid according to the constraint that for all $u, v \in$ W, $(u, v) \notin$ E. The "Number" column means the size of the current set. The "Optimal" column means the rows with the maximum number in this table. The number of the optimal rows can be more than one. We can choose one of them arbitrary. Rows are all possible combinations for vertices in $X_i$ appearing in W. We can use 1 if the corresponding vertex is in W or 0 if the vertex is not in W.

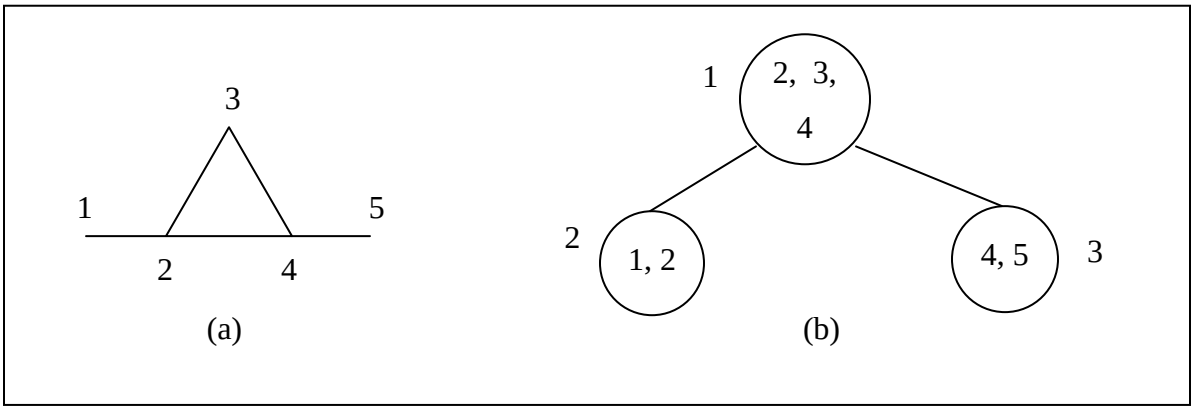


Figure 2.2: A graph and its tree decomposition. The graph (a) has five vertices {1, 2, 3, 4, 5}. Its tree decomposition (b) has treewidth 2 and three tree bags annotated as 1, 2 and 3 separately.

Figure 2.2 shows a graph and its tree decomposition for the maximum independent set problem. Figure 2.3 gives the example of building three tables $\{T_1, T_2, T_3\}$ for the three tree bags

{1, 2, 3} in Figure 2.2 correspondingly. In Figure 2.3, with a postorder traversal of the tree in Figure 2.2(b), we build the table $T_2$ first, the table $T_3$ second and then the table $T_1$ at last. Each row of the tables represents an independent set locally to the tree bags contained in the subtree corresponding to the considered tree node. Computing leaf nodes ($T_2$ and $T_3$) is relatively easy because they do not have children. When computing the table $T_1$, we first compute the validation for each row on its own set $X_1$. We only look up its direct children tables when the combination itself is valid. For example, we look up both $T_2$ and $T_3$ when working on the first, second and third rows of $T_3$. But we do not look up the direct children tables when working on the fourth row of $T_1$ because the vertex 2 and the vertex 3 should not appear in a valid independent set at the same time because they form an edge. And we only look for the local optimal rows according to the overlapped vertices and their values. Choose one arbitrary if there are multiple local optimal rows. For example, when working on the first row of $T_1$, we look up $T_2$ to get the value for the vertex 1 based on the values of overlapped vertices. For this case, the overlapped vertex is the vertex 2 and its value is 0. In $T_2$, there are two valid rows given the value of the vertex 2 is 0. We are only interested in the row with the maximum number, which means we choose the second row where the vertex 1 is 1 because the second row has the set size as 1 compared with the first row where the set size is 0. We choose the global optimal number as our final result once computing the root table is finished. For this example, the size of the maximum independent set is 3. If we need to know what vertices are included in the result, we can use preorder traversal to trace back to get the value for the maximum independent set. We can arbitrarily choose one if there are multiple optimal rows in the root node table. For this example, the maximum independent set is {1, 3, 5}.

| 1 | 2 | 3 | 4 | 5 | Valid | Number | Optimal |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 1 | √ | 2 | |
| 0 | 1 | 0 | 0 | 1 | √ | 2 | |
| 1 | 0 | 1 | 0 | 1 | √ | 3 | √ |
| | 1 | 1 | 0 | | × | | |
| 1 | 0 | 0 | 1 | 0 | √ | 2 | |
| 0 | 1 | 0 | 1 | 0 | × | | |
| | 0 | 1 | 1 | | × | | |
| | 1 | 1 | 1 | | × | | |

(a) $T_1$

| 1 | 2 | Valid | Number | Optimal |
|---|---|---|---|---|
| 0 | 0 | √ | 0 | |
| 1 | 0 | √ | 1 | √ |
| 0 | 1 | √ | 1 | √ |
| 1 | 1 | × | | |

(b) $T_2$

| 4 | 5 | Valid | Number | Optimal |
|---|---|---|---|---|
| 0 | 0 | √ | 0 | |
| 1 | 0 | √ | 1 | √ |
| 0 | 1 | √ | 1 | √ |
| 1 | 1 | × | | |

(c) $T_3$

Figure 2.3: Three tables for the three tree bags in Figure 2.2(b). (a) is the table $T_1$ for the tree bag $X_1$ in Figure 2.2(b), (b) is the table $T_2$ for the tree bag $X_2$ in Figure 2.2(b), and (c) is the table $T_3$ for the tree bag $X_3$ in Figure 2.2(b). Empty cells mean "Not Applicable", in other words, "No need to compute, specify or exist". The dashed cells in (a) are NOT really part of the table, but only for the convenient visualization about the choices of looking up direct children tables.

# CHAPER 3

# BIOPOLYMER FOLDING

A template-based tertiary structure prediction method needs an effective computational algorithm for the alignment between a query biopolymer sequence and an available structure profile. An *ab initio* tertiary structure prediction method requires an effective computational algorithm for fitting a query biopolymer sequence to an unknown 3D conformation. A TDDP may help both tasks. The first supporting evidence is that there are fast approximation algorithms to produce a tree decomposition with decent treewidth performance [5]. The approximation of the treewidth does not affect the optimality but only the running time of the alignment although the bounded treewidth is possibly larger than the minimum treewidth. The second supporting evidence is that the interaction topology graph of a biopolymer tertiary structure usually has a tree decomposition of a small treewidth in nature [14, 28, 29, 35].

The TDDP has been studied for a template-based prediction method for both protein and RNA [14, 28, 29, 35]. However, the power of this method is based on the high quality aligned templates, which are not always available. In addition, the template-based prediction can only predict known structures. As an alternative, the *ab initio* prediction provides a complementary approach for sequences that may have novel folds. We are interested in exploring how the TDDP can be used for the *ab initio* prediction. There are new challenges to apply the TDDP to *ab initio* prediction when considering geometric constraints.

In this chapter, we will describe the basic components of the computational alignment problem, the topology graph, the TDDP solution and challenges for the template-based prediction method. Then we discuss the potential and challenges how the TDDP can be used for the *ab initio* prediction method.

### 3. 1 Template-Based Structure Prediction

The basic idea of a template-based structure prediction is to place the residues of a query biopolymer sequence into structural positions of a template structure in an optimal way governed by a fitness scheme. This procedure is repeated against a collection of templates. The best sequence-structure alignment provides the prediction about the backbone of the query sequence. The fitness scheme is application specific. Usually, a statistical or experimental measurement is used to assess the likelihood of the query sequence fitting into the a structural fold template

The template-based structure prediction usually consists of four components [18, 27]: (1) a query sequence $q$ and a library T, which is a set of templates with 3D biopolymer structures; (2) conformational constraints (e.g. statistical energy functions) for measuring the fitness between the query sequence $q$ and a structure template $t$, where $t \in$ T; (3) an algorithm to align the query sequence $q$ to the template sequence $t$ based on fitness constraints; and (4) a criterion for evaluating the confidence level of the predicted structure. The computational alignment problem exists in the third component.

One particular solution is based on a coarse-grained model [28, 29], which places a stretch of residues as a whole in some structural unit of a structure template. The structural unit in a biopolymer structure is a stretch of continuous residues. In a protein tertiary structure, a structural unit can be a $\alpha$ helix or a $\beta$ strand. For a RNA secondary structure, a structural unit

can be one half of a stem (e.g.: a double helix) formed by a stack of base pairs. A topology graph models each template sequence as a mixed graph, which includes both directed edges and undirected edges. Our framework transforms the sequence-structure alignment to a subgraph isomorphism problem, which is then solved by a TDDP algorithm.

### 3.1.1 Topology Graph Representation and Tree Decomposition

A graph H = (V, E | E = A ∪ D) represents a structure template. A vertex $v$ of V in the Graph H represents a structural unit in the template. In protein tertiary structure templates, a structural unit can be called as a core or core unit. A directed edge in D[H] represents the adjacent connection of the sequence order (e.g.: from N terminal to C terminal) in a biopolymer backbone. An undirected edge in A[H] represents the interaction between two structural units.. Both direct edges D and undirected edges A form the graph edges E. Figure 3.1 gives a protein tertiary structure and its corresponding topology graph.

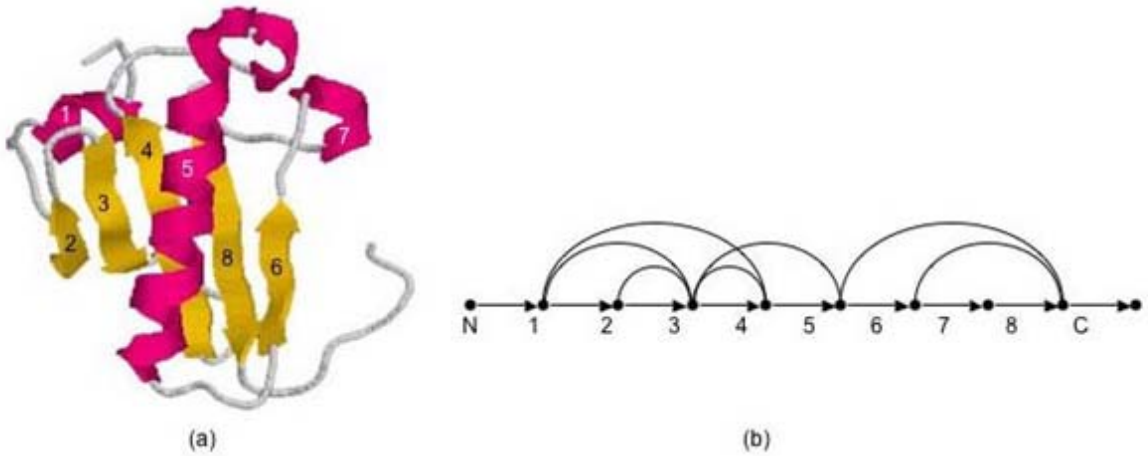


Figure 3.1: A Folded ChainB of Protein Kinase C (PDB-ID 1AV) protein (a) with 8 core units and its corresponding structure graph (b) [29]

A preprocessing component computes the candidates in a query sequence for each core unit of a template sequence for the subgraph isomorphism. Each candidate is represented as a vertex during the preprocessing. A mixed graph G can be constructed in the same way as the template graph H is constructed. The graph G represents the query sequence. The preprocessing results in a map scheme M that suggests the correspondences between the core units in the template sequence and their candidates in the query sequence. The preprocessing usually sets a threshold to choose the top k candidates. The accuracy of the preprocessing plays a key role for the overall sequence-structure alignment.

### 3.1.2 TDDP Algorithm

A tree decomposition (X, T) of a structural graph H represents a tree topology. Figure 3.2 gives the tree decomposition of the graph in Figure 3.1 (b). Based on the tree decomposition, the map

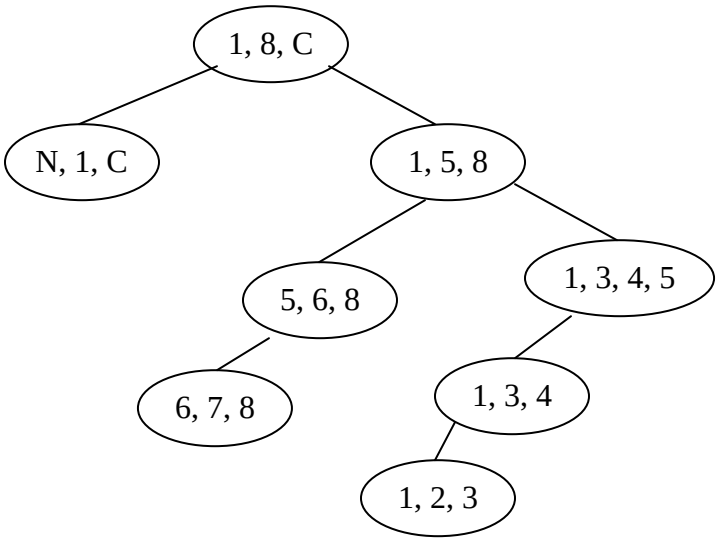


Figure 3.2: A tree decomposition for the structure graph in Figure 3.1(b)

scheme M defined in the preprocessing and a fitness assessment strategy, a bottom up dynamic programming algorithm can be designed along the tree to find the optimal alignment through isomorphism between the template structure graph H and a subgraph of the query sequence graph G.

The algorithm first employs the postorder traversal of the tree to find the optimal alignment value and then uses the preorder traversal of the tree to trace back the actual optimal alignment between each core unit and its corresponding candidate. The postorder traversal process establishes one table for each tree bag and computes every row for local optimal alignment. The internal nodes only need to look up contents in its direct children tables to compute the alignment of the local induced subgraph. Once the computing in the root node is done, a globally optimal alignment can be found and then a trace back process begins to find out the exact alignment.

### 3.1.3 Challenge

The major concerns of efficiently implementing the TDDP algorithm are: (1) the size of the memory space of saving the dynamic tables, (2) the speed of tracing back, and (3) the fitness assessment. If the memory space of storing the dynamic tables is too large, one single commodity computer may not have enough memory to handle the algorithm. It is also critical to keep the tracing back efficient when trying to reduce the space for dynamic tables. The fitness assessment is application specific. We will discuss a strategy corresponding to solve the first two concerns in chapter 4. The strategy can be used without regarding of the type of biopolymers. We will also discuss the energy functions in detail for protein tertiary structure prediction in chapter 4.

### 3. 2 *ab initio* Structure Prediction

The basic idea of a *ab initio* structure prediction method is to search a space of 3D conformations of a query sequence for the most likely conformation which conforms to the secondary and tertiary biological constraints required by the sequence. This method considers only one single query sequence. This method usually consists of three components: (1) a model to derive the space of all possible 3D conformations, (2) an efficient search algorithm to identify the most possible conformation, (3) a criterion for evaluating the confidence level of the predicted structure.

Unlike template-based prediction, *ab initio* prediction considers not only the secondary structure and tertiary interactions but also geometric constraints (e.g.: distance and torsions). The existing methods of modeling all possible 3D conformations tend to be comprehensive in order to include all possible interactions, resulting in computational inefficiency [22]. In particular, an exponential prediction algorithm is usually required to predict the most likely 3D folding. The inefficient computation usually still exists even when adopted a less accurate modeling.

In order to overcome the inefficiency of both modeling and computing, we propose to use the topology graph model and a TDDP algorithm for the *ab initio* prediction. Our methodology is based on two believes. First, a topology graph of a biopolymer structure usually has a small treewidth. Song et al.(2006) found that only 0.8 percent of 3890 proteins tertiary structure templates have tree width $t > 10$ and 92 percent have $t < 6$ when using 7.5 å $C_\beta$ distance cutoff for defining pairwise interactions. A survey [30] on all RNA tertiary structures available in PDB and NDB, about 1581 RNA chains, shows that only 12 of the RNAs have treewidth 5 or larger and more than 99.2% of them have treewidth equals or less than 4. Second, a tree decomposition of the topology graph can be used to partition the geometric space for structure units to reduce

the computing complexity of the prediction. Based on the geometric space partition, we can develop a dynamic programming algorithm to search for the optimal 3D conformation constrained by the secondary and tertiary evolving properties of biopolymers.

In the following, we discuss the topology graph model, space partition, dynamic programming and their challenges for RNA tertiary structure prediction.

### 3. 2.1 Topology Graph Model for RNA Tertiary Structure

The topology graph model is a pair of (G, F), where G = (V, E) and E = A ∪ D. G is a mixed graph. Vertices in V represent some strand regions on the sequence. Non-directed edges in A represent interactions among the strand regions. Directed edges in D represent the connection between two adjacent regions on the backbone. Edges in E consists of edges in A and edges in D. Figure 3.3(b) gives a topology graph of the yeast tRNA (Asp) tertiary structure in Figure 3.3(a). The topology graph is a coarse-grained model. The detailed information related the regions and interactions between them is provided by the function F.

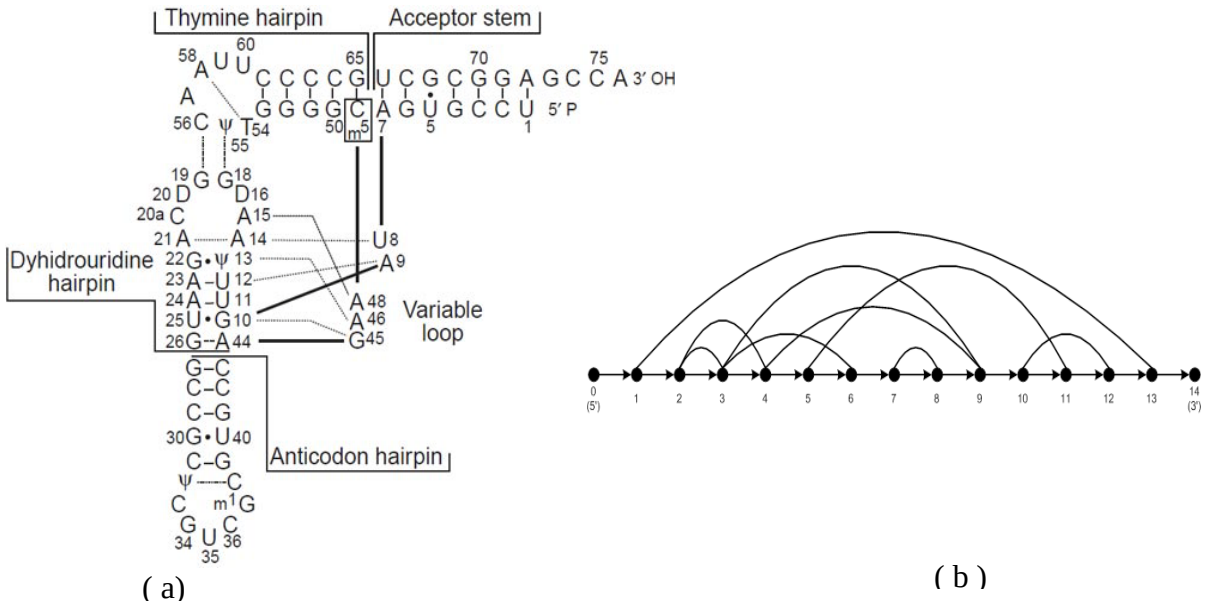


Figure 3.3: The nucleotide interactions of yeast tRNA (Asp) (a) and its corresponding topology graph (b) [22]. The arcs (1,13), (3,6), (7,8) and (10, 12) represent the Acceptor, Dyhidrouridine, Anticodon, and Thymine helices respectively. The rest arcs are for tertiary interaction motifs.

### 3.2.2 Tree Decomposition and Space Partition

We believe there is a relationship between a structure topology graph and its 3D conformations. A tree decomposition of the topology graph can be used to partition the geometric space for structural units. Given a tree decomposition of a topology graph, we propose two steps for the space partition:

(1) Build a 3D sphere for each tree bag;

(2) Merge all 3D spheres together based on their overlapping vertices and biological constraints of a biopolymer

Before implementing the above two steps, we need at least consider a few critical questions as the following. How to decide the diameter of the sphere for each tree bag? How to merge two or more spheres together if they have overlaps? How to merge two or more spheres if they do not have overlaps? How are geometric spaces dynamically assigned and managed to structural units? How to avoid too large designated space for objects which are expected to be close? How to avoid too restricted designated space for the objects to fold? We will discuss strategies and techniques in answering these questions in chapter 5.

### 3.2.3 Dynamic Programming

Given the space partition, we have a designated space for each structural unit. The designated space helps separate the geometric constraints, which results in reducing the alignment computational complexity. A bottom up dynamic programming can carry out to search for the optimal 3D conformation based on their distance, torsions and interactions. A small treewidth of a topology graph in general guarantees that our approach will be practically applicable to large sequences.

# CHAPTER 4

# A MEMORY EFFICIENT ALGORITHM FOR TEMPLATE-BASED PROTEIN TERTIARY STRUCTURE PREDICTION

One of the challengings for implementing the TDDP algorithm for a template-based structure prediction is the memory consumption. The size of the memory usage for saving the dynamic tables grows exponentially as the treewidth. Nowadays, the memory size in a single commodity computer usually is no more than 12GB. It becomes impractical to run the algorithm in a single commodity computer when the memory consumption is over 12GB. If we can decrease the memory consumption by an order of magnitude more, we can handle larger sequences or sequences which have more sophisticated structures. This is important because the treewidth is usually a small number in nature and the number of treewidth grows much slower than the number of residues. The larger treewidth we can handle, the much more biopolymer sequences we can process. This chapter will first propose a heuristic strategy to decrease the size of dynamic tables. Then it will address the computation incorporating energy functions in detail. Finally, it will illustrate the basic components of the TDDP algorithm implementation within the whole architecture.

## 4. 1 Table Size Reduction

We introduce a few notations first as the following: $n$ is the number of core units, $m$ is the number of tree bags, $k$ is the number of candidates for each structure unit, $t$ is the treewidth,

*core* stands for a core unit and *can* stands for a candidate. We explain our work using a simplified example in Figure 4.1. Note that we use numbers to represent the core units in circle and candidates in rectangle, and ignore the amino acid sequences and the loops instead.

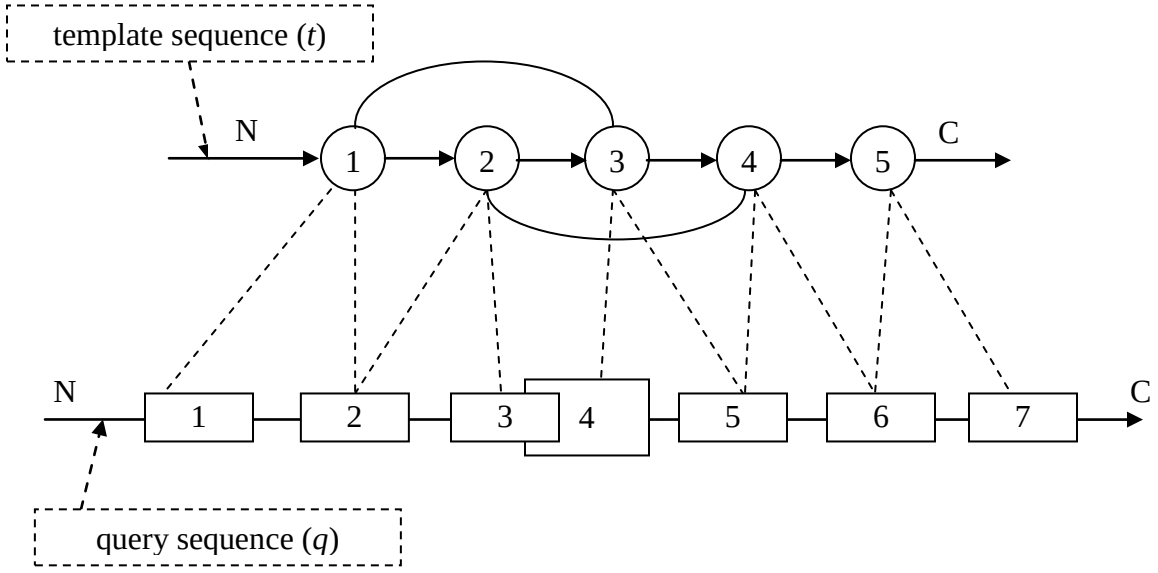


Figure 4.1: The preprocessing result between a query sequence *q* and a template sequence *t*. Assuming each core unit has exactly two candidates (See the relationship between core units and their candidates in Table 4.1). Figure 4.2 gives the constructed template sequence graph. Figure 4.3 gives the tree decomposition with three tree bag labeled as 1, 2, and 3 separately for the graph in Figure 4.3

| Core Unit | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Candidates | 1 | 2 | 4 | 5 | 6 |
| | 2 | 3 | 5 | 6 | 7 |

Table 4.1: The relationship between core units and their candidates

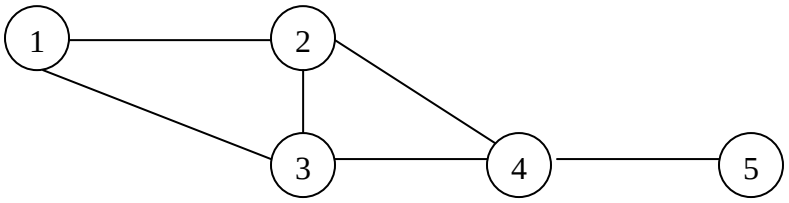


Figure 4.2: The constructed template sequence graph

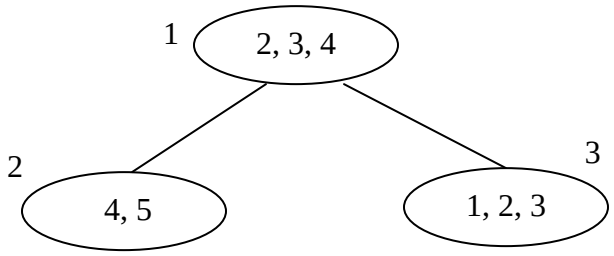


Figure 4.3: The tree generated by a tree decomposition for the graph in Figure 4.2.

A naive structure of a dynamic table for a tree node looks like Table 4.2

| $X_1X_2X_3 \dots X_{t+1}$ | Valid | Score | Optimal |
|---|---|---|---|
| $A_1A_2A_3 \dots A_{t+1}$ | × | | |
| $B_1B_2B_3 \dots B_{t+1}$ | √ | | √ |
| $C_1C_2C_3 \dots C_{t+1}$ | √ | | |

Table 4.2: An abstract basic structure of a dynamic table

The memory space of the table 4.2 is $k^{t+1}(t+1+3)$, where $k^{t+1}$ is the number of rows, t+1 is the number of columns for each core unit and 3 is the number of extra columns for metadata. The "Valid" column records if the combination of the choice is consistent with the sequence position order and non-overlap. The "Score" column records the alignment score calculated based on energy functions. And the "Optimal" column records the optimal choice, which can be either minimum value or maximum value. The decision is defined by the application type. Here,

we use maximization for illustration. Minimization is similar. Note that we can skip saving the table of the root node because the global optimal can be found in this table and there is no need to keep every rows for further reference. We can use only two rows to represent the root node table when computing. One row stores the result of a previous combination. Another row stores the result of a current combination. If the optimal value of the current combination is greater than that of the previous combination, then update the previous result with the current result. Otherwise, keep the previous result as what it is. The procedure is repeated until all combinations are visited. So the following discussion will apply to the tables of non-root nodes.

### 4.1.1 Removal of Redundant Metadata Columns

The first observation on Table 4.2 is that we can combine the three extra metadata columns into one. We can remove the "Valid" column if we use 0 or a negative number (e.g.: -1) in the "Score" column to mark the invalid rows. We can easily search the optimal row based on the value in the "Score" column. So we can remove the "Optimal" column as well. As a result, we only need keep the "Score" column. Then the memory space of the table becomes $k^{t+1}(t+1+1)$. Table 4.3 gives the dynamic table for the tree bag 2 in Figure 4.3.

### 4.1.2 Breakdown of Dynamic Tables

The second observation is that we need a good strategy to improve the speed of tracing back during the dynamic programming process. In Figure 4.3, the overlapping vertices between tree bag 2 and its parent tree bag 1 are $\{2,3\}$. When building the dynamic table of the parent, we look up its direct children tables to get the corresponding optimal value based on the identified values for the overlapped part (2,3). Obviously, it can be computationally expensive to search through

the whole table of the tree bag 2 to look up a specific row or a few related rows. We need a map function to calculate the row in a table for a given combination of values on part of the vertices. The desired asymptotic looking up time should be O(1) with hidden constant factor as small as possible.

| Core Unit | 1 | 2 | 3 | Score |
|---|---|---|---|---|
| Candidate | 1 | 2 | 4 | 4.5 |
| | 2 | 2 | 4 | 0 |
| | 1 | 3 | 4 | 0 |
| | 2 | 3 | 4 | 0 |
| | 1 | 2 | 5 | 4.8 |
| | 2 | 2 | 5 | 0 |
| | 1 | 3 | 5 | 5.1 |
| | 2 | 3 | 5 | 5.4 |

Table 4.3: The dynamic table with only the “Score” column. We use 0 to specify the invalid rows.

In order to generate a map function we can store the data in a consistent way. One good way is to sort the data before storage. Here we store the data in an ascending order for each column from right to left. We also sort the vertices by their orders in a template sequence in an ascending order for each dynamic table. See Table 4.3 as a reference.

Given the combinatorial values of the overlapped vertices, there may be more than one row in one of its direct children tables which meet the optimal requirement. For example, the first row and the second row in Table 4.3 meet the requirement that core unit 2 is mapped to candidate 2 and core unit 3 is mapped to candidate 4. Obviously, it is more efficient if we can

calculate the starting row only once and then conduct a sequential search for other related rows. This motivates us to keep the overlapped columns together at the right most part of the table.

After we locate the corresponding rows in one of its children tables, we need to choose an optimal row. In other words, the parent node only cares about the optimal combination in its direct child tables. It seems unnecessary to sort the multiple values and then choose the optimal value during the looking up process. This property motivates us to continue to refine the data storage to remove the redundant computation of sorting. One good way is to break the table into two sub-tables. One sub-table, called the "overlapped table" (see Figure 4.4 (a)), is constructed for the overlapped vertices. The "Score" column in this table only records the optimal value. And the rest of vertices goes into another sub-table, called the "non-overlapped table" (see Figure 4.4 (b)).

| Core Unit | 2 | 3 | Score | Pointer |
|---|---|---|---|---|
| Candidate | 2 | 4 | 4.5 | 1 |
| | 3 | 4 | 0 | 0 |
| | 2 | 5 | 4.8 | 1 |
| | 3 | 5 | 5.4 | 2 |

| 1 | Core Unit |
|---|---|
| 1 | Candidate |
| 2 | |

(a) Overlapped table (b) Non-overlapped table

Figure 4.4: The breakdown of a dynamic table. (a) is the overlapped table. (b) is the non-overlapped table. The overlapped table has an extra "Pointer" column which records which row in the non-overlapped table where the optimal score is chosen. If the row in the overlapped table is invalid, we can simply ignore the value of the pointer and set it as a 0.

The new arranged data structure allows us to only calculate the corresponding row during the looking up process. The "Pointer" column is very important because we also need to know the value for the non-overlapped vertices when tracing back from the root. We use $ol$ stands for the number of overlapped vertices between the current tree bag and its parent tree bag. Then the memory space for one dynamic table becomes

$$k^{ol}(ol+2)+ k^{t+1-ol}(t+1-ol)$$

In a worst case, where $ol = t$ or $ol = 1$, the memory space is $k^t(t+2)+k$. We reduce the memory space by almost 1/*k*. In a roughly best case, where $ol$ = ($t$+1)/2, the memory space is $k^{(t+1)/2}(t+1+2)$. We reduce the memory space by almost $1/\ k^{(t+1)/2}$

According to the sorted data structure, we come up with the row number prediction equation below given the column values

$$\text{Row Number} = \sum_{i\in overlap}[(q_i - 1) * k^{i-1} + 1] \ - |overlap| + 1 \qquad (4.1)$$

where $overlap$ is an integer set of the overlapped columns, which starts with 1, $i$ is the number of the specific column, $k$ is the number of candidates and $q_i$ is the position of the selected candidate in the candidate pool of the corresponding core unit. We take Table 4.4(a) as an example. Calculate the row number when core2 = can3 and core3 = can4. First, we know k = 2. Given core2 = can3, we get $i$ = 1 and $q_1$ = 2 (because can3 is the second choice). So the first row number given core2 = can3 can be calculated as equation 4.2

$$\text{Row Number} = 1 * 2^0 + 1 = 2 \qquad (4.2)$$

Then given core3 = can4, we get $i$ = 2 and $q_2$ = 1. So the first row number given core3 = can4 can be calculated as equation 4.3

$$\text{Row Number} = 0 * 2^1 + 1 = 1 \qquad (4.3)$$

We can get the final result by combing the equation 4.2 and equation 4.3 according to equation 4.1 as: Row Number = 2 + 1-2 +1= 2.

### 4.1.3 Space Compression

The third observation is that keeping all of the concrete combinations of the candidates in a table is redundant. We can use one unique number $i$ to represent each row and then we do not need to keep the concrete content $\{A_1A_2A_3 \dots A_{t+1}\}$ in the row any more if we have a map function computing the relationship between the unique number $i$ and its content. The map function can be described as:

$$Map\ (i) \rightarrow \{A_1A_2A_3 \dots A_{t+1}\}$$

If we use an array to store the score value, then we can use the position of each score as a key mapping to its content. Following the logic, we convert the two tables in Figure 4.4 to Table 4.4. For example, the first row in Table 4.4 maps to the first row in Figure 4.4 (a). We can simply describe the map function as Map(1) = {2, 4}. At the same time, the value 1 in the pointer column of the first row in Table 4.1 can be mapped into the first row of Figure 4.4 (b) as Map(1) = {1}.

| Score | Pointer |
|---|---|
| 4.5 | 1 |
| 0 | 0 |
| 4.8 | 1 |
| 5.4 | 2 |

Table 4.4: The final data structure of a dynamic table

Now, the memory space complexity is reduced to $2k^{ol}$. In a worst case, where $ol = t$, the memory space complexity is $2k^{t}$. In a best case, where $ol = 1$, the memory space complexity is 2k. In a roughly average case, where ol = ($t$+1)/2, the memory space complexity is $2k^{(t+1)/2}$, which reduces the memory usage from the original naïve approach $k^{t+1}(t+1+3)$ by $1/k^{(t+1)/2}$.

Here, we give the map function for each column. Given the row number $i$ and the column position $q$, and the candidate pool, which can be described as an array A, the value can be calculated as

$$\text{Value} = A[p],\ b = \left\lceil \frac{i \bmod k^{q}}{k^{q-1}} \right\rceil \text{ and } p = \begin{cases} b, & if\ b \neq 0 \\ length(A), & if\ b = 0 \end{cases}$$

where the $mod$ represents the modulo operation, and $p$ represents the index position in the array A, which stores the value of candidates of a core unit. We use Figure 4.4 (a) as an example. Given the row number 3, predict its content as the following. For column 1, $q$ = 1, $k$ = 2, A = {2, 3}. We get $p$=$b$= $\left\lceil \frac{3 \,\%\, 2^{1}}{2^{1-1}} \right\rceil$ = 1. So the value for the first column will be A[1] = 2. For column 2, q = 2, k = 2, A = {4, 5}, we get $p$=$b$=$\left\lceil \frac{3 \,\%\, 2^{2}}{2^{2-1}} \right\rceil$ = 2. So the value for the second column will be A[2] = 5. Putting the above result together, we calculate the values of the third row as {2, 5}, which is the same as the third row in Figure 4.4(a).

We give a more concrete example below in terms of how much memory space is saved. We assume 4B to represent each column, $k = 10$ and $t$ = 9. The size for the biggest table under a naive approach will be: $10^{10} \times (10+3) \times 4\text{B}$ = 520GB. In a worst case where $ol = 9$, we calculate the size of the table as $10^{9} \times 2 \times 4\text{B}$ = 8GB, which saves 440GB memory space. In a best case, where $ol$ = 1, we calculate the size of the table as $10^{1} \times 2 \times 4\text{B}$ = 80B, which saves

519.99999992GB memory space. In a roughly average case, where $ol = 5$, we calculate the size of the table as $10^5 \times 2 \times 4B = 800KB$, which saves 519.9992GB memory space.

### 4. 2. Energy Functions

For template-based protein tertiary structure prediction, energy functions are critical to both the preprocessing step and the TDDP step. The preprocessing step uses energy functions to choose strong candidates. The TDDP step uses the energy functions to identify the optimal combination. The energy functions used in both steps have some overlaps. It is very important to keep the overlapped functions consistent. We will focus on the energy functions used in the TDDP step. We refer the reader to [18] for energy functions used in the preprocessing step.

In the following, we will first define some terminologies. Then we will use a simplified example to explain the calculation with energy functions.

#### 4. 2.1 Terminologies

Let M stands for the function to calculate mutation energy, S stands for the function calculate singleton energy, G stands for the function to calculate gap penalty, SS stands for the function to calculate secondary structure match and P stands for the function to calculate pairwise interaction energy. $core$ stands for a Core Unit (e.g.: α helix or β strand) in the template sequence. $can$ stands for a candidate in a query sequence for a corresponding core unit

**4. 2.2 A Simplified Example**

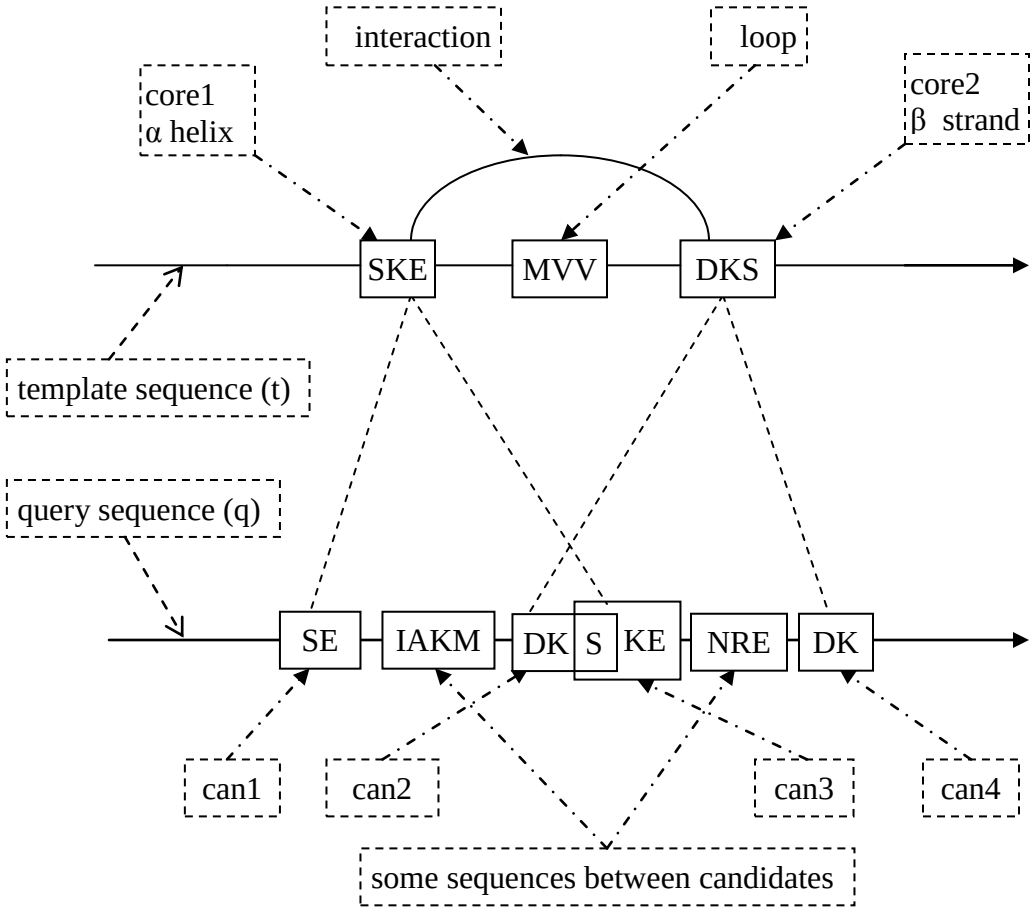


Figure 4.5: A simplified example of an alignment between a template sequence *t* and a query sequence *q*. In the template sequence, SKE is a α helix annotated as core1, MVV is a loop, and DKS is a β strand annotated as core2. Each core unit in the template has exactly two candidates in the query sequence. In the query sequence *q*, there are total 4 different candidate blocks, which are annotated as can1, can2, can3 and can4 separately.

Figure 4.5 gives a template sequence *t*, a query sequence *q* and their alignment. The template sequence consists of SKEMVVDKS. The block SKE from the first residue to the third residue forms a $\alpha$ helix, which is annotated as core1. The block DKS from the seventh residue to the ninth residue forms a β strand, which is annotated as core2. The block MVV between core1 and core2 is a loop. There are interaction between core1 and core2. The query sequence consists of SEIAKMDKSKENREDK. Table 4.5 lists the four alignments between core units and their candidates. Table 4.6 gives the interaction information for the template sequence. We ignore the interaction between cores and loops. Table 4.7 gives all of the combinations of aligning core units to their candidates.

| Core Unit | SKE (core1) | SKE (core1) | DKS (core2) | DKS (core2) |
|---|---|---|---|---|
| Corresponding Candidate | S-E (can1) | SKE (can2) | DKS (can3) | DK- (can4) |

Table 4.5 the alignment tables. "-"means gap

| Residue Number | Residue Number |
|---|---|
| 1 | 7 |
| 3 | 8 |
| 3 | 9 |

Table 4.6: The two body interactions of the template sequence *t*

| core1 | core2 | Valid |
|---|---|---|
| can1 | can2 | Yes |
| can3 | can2 | No (because of overlap and wrong order) |
| can1 | can4 | Yes |
| can2 | can4 | yes |

Table 4.7: All combinations of mappings between core units and their candidates

### 4. 2. 3 Types of Energy Functions

There are three energy functions, together defining the objective function to find the minimum global energy value by searching all of the combinations.

1) Energy function S1. This is the alignment score between a core and its candidate.

2) Energy function S2. This is the alignment score between a loop in a template sequence and a corresponding loop in a query sequence.

3) Energy Function S3. This is the pairwise interaction value.

### 4.2.4 Calculation of Energy Functions

In order to calculate the objective function, we need to calculate the individual energy function for mutation energy, singleton energy, gap penalty, secondary structure match and pairwise interaction energy. We adopt the similar techniques used at [18].

In addition, we also need to define a weight for each energy value. The weight is sensitive to the dataset used and usually tuned according to the specific dataset. Here we adapt the same weights used in the work of [18]. Table 4.8 gives the values.

| Type of energy | Weights | Notation |
|---|---|---|
| Mutation | 0.002964 | $w_m$ |
| Singleton | 0.892100 | $w_s$ |
| Gap Penalty | 0.036929 | $w_g$ |
| Secondary Structure Match | 0.312064 | $w_{ss}$ |
| Pairwise Interaction | 0.324656 | $w_p$ |

Table 4.8: Weights on different type of energies

We take the combination of the first row in Table 4.7 as an example to illustrate the calculation of the three functions. The energy function S1 scores the alignment between a core and its candidate. The function value should be produced in the preprocessing step. We omit the detail here and refer the reader to [18] for more details. For the discussed example, the energy calculation can be formulated with the equation 4.4

S1 = $w_m$ (M(core1, can1) + M(core2, can2)) + $w_g$ (G(core1, can1) + G(core2, can2)) + $w_s$(S(core1, can1) + S(core2, can2)) + $w_{ss}$(SS(core1, can1) + SS(core2, can2)) (4.4)

The energy function S2 scores the alignment between a loop in a template sequence and a corresponding loop in a query sequence. In our example, the loop in the template sequence is MVV and the loop in the query sequence is IAKM. We can use Needleman-Wunsch algorithm [9] to align the query loop and the template loop. The recurrence for the alignment score is the following

$$\text{F (i, j)} = \max \begin{cases} F(i-1, j-1) + s(x_i, y_j), & \\ F(k, j) - g(i-k), & k = 0, \dots, i-1, \\ F(i, k) - g(j-k), & k = 0, \dots, j-1. \end{cases} \quad (4.5)$$

where $x_i$ represents the $i$th amino acid of the template loop, $y_j$ represents the $j$th amino acid of the query loop, F(i, j) is the best alignment score between the initial segment $x_{1\dots i}$ of x up to $x_i$

and the initial segment $y_{1\ldots j}$ of y up to $y_j$, and $g(d)$ is the affine gap penalty score, which can be calculated with the equation 4.6

$$g(d) = \gamma + \delta\,(|d|\text{-}1) \tag{4.6}$$

where $\gamma$ is the gap opening penalty, $\delta$ is the gap extension penalty, and $d$ is the number of gaps. In our case, we set $\gamma$ as 10.8 and $\delta$ as 0.6. $s(x_i, y_j)$ is the sum of related individual energies, which can be calculated with the equation 4.7

$$s(x_i, y_j) = w_m \mathrm{M}\,(x_i, y_j) + w_s \mathrm{S}\,(x_i, y_j) + w_{ss} \mathrm{SS}(x_i, y_j) \tag{4.7}$$

$\mathrm{M}(x_i, y_j)$ is the mutation energy, whose detail can be referred to the mutation matrix Table 4.9. $\mathrm{S}(x_i, y_j)$ is the singleton energy, which evaluates the preference of the place of every amino acid in a residue location within a core of certain solvation accessibility. $\mathrm{SS}(x_i, y_j)$ is the secondary structure energy, whose detail can be referred to the secondary structure scoring matrix in Table 4.10.

A noteworthy point here is that the dynamic programming based on the object function described in the equation 4.5 will have the running time $\mathrm{O}(n^3)$, which is OK for short sequences alignment but may be a problem for longer sequences. To reduce the running time from $\mathrm{O}(n^3)$ to $\mathrm{O}(n^2)$, a more sophisticated recurrence than the equation 4.5 would be needed. We can modify the object function by adopting a simplified gap function

$$g(d) = \mathrm{d} * \gamma \tag{4.8}$$

where $d$ is the number of gaps and $\gamma$ is the single gap penalty. Then the new object function becomes equation 4.9

$$\mathrm{F\ (i,\ j)} = \max \begin{cases} F(i-1, j-1) + s(x_i, y_j), \\ F(i-1, j) - g(1), \\ F(i, j-1) - g(1), \end{cases} \tag{4.9}$$

| | A | R | N | D | C | Q | E | G | H | I | L | K | M | F | P | S | T | W | Y | V |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | 2 | | | | | | | | | | | | | | | | | | | |
| R | -2 | 6 | | | | | | | | | | | | | | | | | | |
| N | 0 | 0 | 2 | | | | | | | | | | | | | | | | | |
| D | 0 | -1 | 2 | 4 | | | | | | | | | | | | | | | | |
| C | -2 | -4 | -4 | -5 | 12 | | | | | | | | | | | | | | | |
| Q | 0 | 1 | 1 | 2 | -5 | 4 | | | | | | | | | | | | | | |
| E | 0 | -1 | 1 | 3 | -5 | 2 | 4 | | | | | | | | | | | | | |
| G | 1 | -3 | 0 | 1 | -3 | -1 | 0 | 5 | | | | | | | | | | | | |
| H | -1 | 2 | 2 | 1 | -3 | 3 | 1 | -2 | 6 | | | | | | | | | | | |
| I | -1 | -2 | -2 | -2 | -2 | -2 | -2 | -3 | -2 | 5 | | | | | | | | | | |
| L | -2 | -3 | -3 | -4 | -6 | -2 | -3 | -4 | -2 | 2 | 6 | | | | | | | | | |
| K | -1 | 3 | 1 | 0 | -5 | 1 | 0 | -2 | 0 | -2 | -3 | 5 | | | | | | | | |
| M | -1 | 0 | -2 | -3 | -5 | -1 | -2 | -3 | -2 | 2 | 4 | 0 | 6 | | | | | | | |
| F | -4 | -4 | -4 | -6 | -4 | -5 | -5 | -5 | -2 | 1 | 2 | -5 | 0 | 9 | | | | | | |
| P | 1 | 0 | -1 | -1 | -3 | 0 | -1 | -1 | 0 | -2 | -3 | -1 | -2 | -5 | 6 | | | | | |
| S | 1 | 0 | 1 | 0 | 0 | -1 | 0 | 1 | -1 | -1 | -3 | 0 | -2 | -3 | 1 | 2 | | | | |
| T | 1 | -1 | 0 | 0 | -2 | -1 | 0 | 0 | -1 | 0 | -2 | 0 | -1 | -3 | 0 | 1 | 3 | | | |
| W | -6 | 2 | -4 | -7 | -8 | -5 | -7 | -7 | -3 | -5 | -2 | -3 | -4 | 0 | -6 | -2 | -5 | 17 | | |
| Y | -3 | -4 | -2 | -4 | 0 | -4 | -4 | -5 | 0 | -1 | -1 | -4 | -2 | 7 | -5 | -3 | -3 | 0 | 10 | |
| V | 0 | -2 | -2 | -2 | -2 | -2 | -2 | -1 | -2 | 4 | 2 | -2 | 2 | -1 | -1 | -1 | 0 | -6 | -2 | 4 |

Table 4.9: Mutation energy matrix [18]

| | $\alpha$ | | | $\beta$ | | | Loop | | |
|---|---|---|---|---|---|---|---|---|---|
| | buried | intermediate | exposed | buried | intermediate | exposed | buried | intermediate | exposed |
| A | -0.627 | -0.122 | -0.070 | -0.026 | 0.699 | 1.148 | -0.056 | 0.248 | 0.430 |
| R | 0.935 | -0.595 | -0.579 | 1.188 | -0.445 | -0.663 | 0.924 | -0.068 | -0.085 |
| N | 0.858 | 0.213 | -0.072 | 0.851 | 0.259 | -0.083 | 0.089 | -0.280 | -0.571 |
| D | 1.155 | 0.198 | -0.454 | 1.108 | 0.310 | -0.086 | 0.351 | -0.179 | -0.652 |
| C | -0.383 | 0.600 | 2.421 | -0.614 | 0.278 | 1.628 | -0.727 | -0.068 | 1.522 |
| Q | 0.652 | -0.433 | -0.796 | 0.980 | -0.056 | -0.550 | 0.851 | 0.154 | -0.219 |
| E | 1.017 | -0.344 | -1.079 | 1.404 | 0.072 | -0.749 | 1.326 | 0.485 | -0.372 |
| G | 0.477 | 0.995 | 1.406 | 0.212 | 0.559 | 1.460 | -0.483 | -0.407 | -0.481 |
| H | 0.432 | -0.210 | 0.219 | 0.245 | -0.326 | 0.023 | -0.031 | -0.266 | 0.016 |
| I | -0.559 | 0.185 | 1.456 | -0.852 | -0.114 | 0.641 | -0.159 | 0.541 | 1.506 |
| L | -0.721 | -0.162 | 1.000 | -0.388 | 0.263 | 0.908 | -0.224 | 0.241 | 1.275 |
| K | 1.645 | -0.312 | -0.895 | 1.826 | -0.254 | -0.889 | 1.891 | 0.236 | -0.507 |
| M | -0.656 | -0.158 | 0.992 | -0.250 | 0.245 | 0.980 | -0.240 | 0.191 | 0.762 |
| F | -0.470 | 0.229 | 1.490 | -0.601 | -0.086 | 1.044 | -0.438 | 0.093 | 1.379 |
| P | 1.181 | 0.755 | 0.497 | 1.173 | 0.729 | 0.281 | -0.430 | -0.573 | -0.559 |
| S | 0.359 | 0.346 | 0.050 | 0.303 | 0.056 | -0.111 | -0.145 | -0.205 | -0.270 |
| T | 0.256 | 0.180 | 0.356 | 0.088 | -0.403 | -0.574 | -0.011 | -0.096 | -0.067 |
| W | -0.374 | -0.197 | 1.533 | -0.295 | -0.455 | 0.821 | -0.338 | -0.047 | 1.312 |
| Y | -0.100 | -0.233 | 0.950 | -0.293 | -0.619 | 0.228 | -0.051 | -0.153 | 0.927 |
| V | -0.370 | 0.270 | 1.365 | -0.893 | -0.317 | 0.322 | -0.039 | 0.406 | 1.208 |

Table 4.10: Secondary structure scoring matrix [18]

| | A | R | N | D | C | Q | E | G | H | I | L | K | M | F | P | S | T | W | Y | V |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | -453 | | | | | | | | | | | | | | | | | | | |
| R | 171 | 427 | | | | | | | | | | | | | | | | | | |
| N | 264 | 344 | -89 | | | | | | | | | | | | | | | | | |
| D | 377 | 32 | 211 | 706 | | | | | | | | | | | | | | | | |
| C | -361 | 66 | 39 | 336 | -2386 | | | | | | | | | | | | | | | |
| Q | 220 | 316 | 203 | 410 | 144 | 212 | | | | | | | | | | | | | | |
| E | 391 | 5 | 425 | 921 | 448 | 434 | 790 | | | | | | | | | | | | | |
| G | -81 | 200 | 41 | 244 | -278 | 265 | 524 | -246 | | | | | | | | | | | | |
| H | 94 | 249 | 192 | 122 | -367 | 250 | 340 | 38 | -266 | | | | | | | | | | | |
| I | -523 | 149 | 222 | 443 | -591 | 180 | 362 | 6 | -46 | -940 | | | | | | | | | | |
| L | -488 | 83 | 341 | 540 | -512 | 137 | 386 | 101 | -9 | -698 | -698 | | | | | | | | | |
| K | 396 | 846 | 474 | 135 | 314 | 433 | 53 | 449 | 549 | 275 | 368 | 725 | | | | | | | | |
| M | -342 | 152 | 166 | 432 | -576 | 193 | 441 | -64 | -104 | -514 | -434 | 402 | -732 | | | | | | | |
| F | -302 | 143 | 200 | 474 | -554 | 184 | 485 | -71 | -91 | -576 | -515 | 384 | -539 | -601 | | | | | | |
| P | 145 | 359 | 289 | 496 | -159 | 346 | 546 | 108 | 148 | 92 | 126 | 679 | 25 | -0 | 124 | | | | | |
| S | 91 | 251 | 55 | 228 | -194 | 216 | 374 | -6 | 32 | 53 | 125 | 397 | 46 | 33 | 198 | -14 | | | | |
| T | -46 | 238 | 20 | 252 | -213 | 155 | 360 | -49 | 7 | -196 | -49 | 343 | -37 | -1 | 165 | -13 | -161 | | | |
| W | -165 | -55 | 63 | 424 | -517 | 53 | 507 | -83 | -163 | -348 | -353 | 263 | -396 | -462 | -234 | 58 | 46 | -604 | | |
| Y | -202 | -43 | 115 | 398 | -472 | 123 | 438 | -74 | -84 | -496 | -408 | 101 | -410 | -451 | -117 | 75 | 11 | -433 | -471 | |
| V | -504 | 104 | 230 | 422 | -612 | 174 | 371 | -42 | -53 | -886 | -721 | 297 | -497 | -538 | 48 | 45 | -222 | -320 | -444 | -946 |

Table 4.11: The two body interaction matrix [18]

We believe the object function 4.5 is more accurate than the object function 4.9. We also foresee the length of the loop segment sequences will be short for most cases. That means the O($n^3$) running time could be feasible. But for some long loop sequences, we may have to either adopt a O($n^2$) running time algorithm or look for more insights to add some boundary limitations to reduce the equation 4.5 to a O($n^2$) running time algorithm.

The energy function S3 computes the pairwise interaction contribution. Usually, the template profile gives the pairs of residue-residue interactions. We also know the alignment between a core and its candidate. Then we can use the residues in the candidate to replace the residues in the core unit to calculate the pairwise interaction energy based on the two body interaction matrix in Table 4.11. In our example, we get the alignments (SKE, S-E) and (DKS, DKS). Based on the residue–residue interactions listed in Table 4.6, we can calculate the pairwise interaction energy values as the following:

P((core1, can1), (core2, can2)) = P(S,D)+P(E,K)+P(E,S)

### 4. 3 Overview of the TDDP Algorithm Implementation

A template-based structure prediction is trying to align a query sequence to a collection of templates and identify the most optimal alignment. Here we only focus on one query sequence and one template sequence since the repeated alignment process follows the same logic.

First, the alignment takes the following input: a query sequence, a template sequence, energy functions and the preprocessing result between the template sequence and the query sequence. The template sequence gives information about the core units and their start positions and end positions, and the interactions among core units. The energy functions and the related scoring matrices specify how to calculate the three alignment fitness scores. The preprocessing result gives the pairs of each core and its candidates with starting positions and end positions in the query sequence. The preprocessing result also specifies the alignments between each core unit and its candidates and the corresponding alignment scores.

Second, it builds a topology graph based on the secondary structure of the template. It annotates the core unit using integer numbers starting with 1 following the order along the biopolymer backbone direction. It then runs a tree decomposition program to obtain a tree topology of the sequence graph.

Third, it builds a matrix for the preprocessing result. Each column is corresponding to one core unit in the template sequence. For each column, the rows store its candidates. The candidates should be sorted by their order in the query sequence. The matrix is used to compute the content in the dynamic tables later on.

Fourth, it builds three matrices for each energy function. The major motivation of computing the energy function matrices in advance is to reduce the duplicated calculation when

calculating the fitness score for valid combinations. As a consequence, the computed energy matrices will speed up the dynamic programming process.

Fifth, it walks though the tree by postorder traversal to compute the optimal alignment. For each tree node, it creates two indexes for the two temporary tables (see Figure 4.4 as an example). One is for the overlapped vertices with its parent node and another is for the rest non-overlapped vertices. Then it creates the dynamic table (see Table 4.4 as an example) for this tree bag to record the score and pointer values by computing all combinations. It doesn't create a table for the root node as we discussed in 4.1. Note that duplicate calculations on the internal nodes should be avoided because some energy functions for the overlapped vertices with its direct children tables may already be computed.

Finally, it traverses the tree by preorder to obtain the optimal alignment. It outputs the optimal alignment score and the pairs between core and its candidate.

# CHAPTER 5

## A TOPOLOGY MODEL FOR *ab initio* TERTIARY STRUCTURE PREDICTION

We briefly introduced a topology graph model, a space partition strategy and a TDDP algorithm for *ab initio* prediction in chapter 3. To be more precise, we define the following five steps that carry out the prediction process: (1) derive a topology graph for a given query sequence, (2) build a sphere for each tree bag based on the tree decomposition of the topology graph, (3) partition the conformational space by merging spheres, (4) generate discrete candidates for each vertex in its own designated space, (5) search for the most likely optimal folding based a bottom up dynamic programming. This chapter, as part of an ongoing project, mainly focuses on step 3, step 4 and step 5. We will use RNA 3D folding as an example. The framework will be applicable to *ab initio* protein tertiary structure prediction although RNAs and proteins have different biological properties.

Given a topology graph model G, we can get a tree decomposition T of this graph with a bounded treewidth *k*. Based on the tree decomposition, we can then build one 3D sphere $S_i$ = ($C_i$, $D_i$ ) for each tree bag $X_i$, where $C_i$ is the origin of this sphere and $D_i$ is the diameter of this sphere. Each 3D sphere $S_i$ has an associated biological constraint $Z_i$ , let $\bigcup_{i \in I} Z_i$ = F, which is the biological constraints of a biopolymer. The diameter of a 3D sphere can be initially decided by the minimum length to lay down all of the nucleotides coaxially based their relationships. We define D as the diameter calculation function. Given a tree bag $X_i$, the $D_i$ can be calculated with the equation 5.1

$$D_i = \mathrm{D}\,(X_i, F) \tag{5.1}$$

**5.1 Space Partition**

Given the 3D spheres, the tree decomposition T and the biological constraint F, we can merge the spheres based on their overlapped vertices. Each vertex will have its own designated restricted area once the merging is done. The restricted area serves for the purpose of decreasing the computational alignment between a query sequence and its possible 3D conformations. The real challenge here is how to consistently merge spheres so that the space partition can serve well for candidate generation and a following optimal search under the constraint F. There are four basic factors to be considered when merging spheres: (1) merging order and coordinate system, (2) distance between spheres, (3) torsion, and (4) space redistribution for each vertex.

1) Merging order and coordinate system. Each sphere has its own coordinate system originally. It is better to merge spheres together based on one common coordinate system. Using one common coordinate system can reduce the transform computing caused by different coordinate systems. When a new sphere is merged with the existed merged space, it follows the same common coordinate system. We propose a top down approach to carry out the merging process. The coordinate system of the root tree bag serves for the one common coordinate system. Then its direct children, if existed, added. The procedure continues until all nodes are visited.

2) Distance between overlapped spheres. We define the overlapped vertices $W_{ij} = X_i \cap X_j$ for merging $S_i$ and $S_j$. We then define the distance between $S_i$ and $S_i$ with the equation 5.2

$$\mathrm{D}(S_i, S_j) = (D_i + D_j)/2 - \mathrm{D}(W_{ij}, F) \tag{5.2}$$

Figure 5.1 gives an example about merging two spheres.

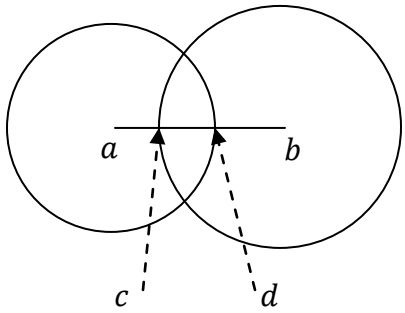


Figure 5.1: Distance between two overlapped spheres. Point $a$ is the center of the left sphere $S_i$ , and point $b$ is the center of the right sphere $S_j$. Point $c$ and pointer $d$ are the intersecting points between the line $(a, b)$ and the surfaces of the two spheres. The distance between $c$ and $d$ is D($W_{ij}, F$). The distance between $a$ and $b$ is D($S_i$, $S_j$) .

3) Torsion. Deciding the torsion is the most critical part for the space partition. Bad torsion choice may predict an improperly restricted space. Too small space may not hold the vertices which are supposed to have some minimum distance restriction. Too large space may obstruct the vertices in other spaces to be physically close enough to meet the biological constraint F. We propose a conservative parameter based approach. When merging a sphere with the existing merged spheres, we move them close enough but do not violate the constraint F. We conservatively leave maximum open space for future merging. We can use a fixed set of angles for merging spheres. For example, {60, -60, 180} could be such a kind of set. We can then develop rules that describe the mapping between the selection of the angle value and its scenario based on future experiments of known RNA tertiary structures.

4) Space redistribution. We define two types of regions for the overlapped spaces between spheres. One is called as intended overlapped region, which holds the overlapped vertices. Another is called as unintended overlapped region, which doesn't hold the overlapped

vertices but is part of the intersection between spheres. Figure 5.2 gives a tree decomposition for the graph in Figure 3.3(b). Figure 5.3 gives an example of merging spheres of tree bag 3, tree bag 5 and tree bag 7 in Figure 5.2. The merging in Figure 5.3 mainly focuses on illustrating the

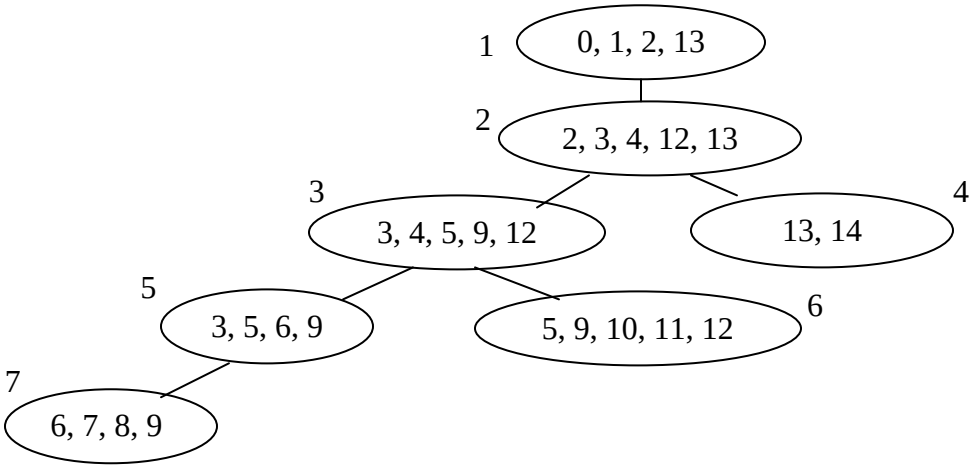


Figure 5.2: A tree decomposition of the graph in Figure 3.3(b) with 7 tree bags

related concepts in space redistribution and doesn't necessary follow up the top down merging order. We define the $i$th isolated region as $R_i$ caused by merging. Figure 5.2(a) is about merging $S_3$ and $S_5$. Figure 5.2(b) gives an example annotating the isolated regions based on the merging in Figure 5.2(a). For example, $R_2$ in Figure 5.2(b) is the designated space for vertices {3, 5, 9} in Figure 5.2(a). $R_2$ in Figure 5.2(b) is an intended overlapped region. Figure 5.2(c) adds the sphere $S_7$. Figure 5.2(d) gives the new updated isolated regions and their corresponding annotations. Now $R_2$ in Figure 5.2 (b) is split into two sub-regions: $R_2$ and $R_5$ in Figure 5.2(d), where $R_2$ covers the vertices {3, 5} and $R_5$ covers the vertex {9}. In Figure 5.2(d), $R_1$, $R_2$ and $R_5$ are intended overlapped regions, but $R_6$ is an unintended overlapped region because there is no explicitly associated vertex. For an unintended overlapped region, we adopt a "first occupied, first use" principle. $R_6$ in Figure 5.2(d) is part of $R_3$ in Figure 5.2(b). We know this $R_3$

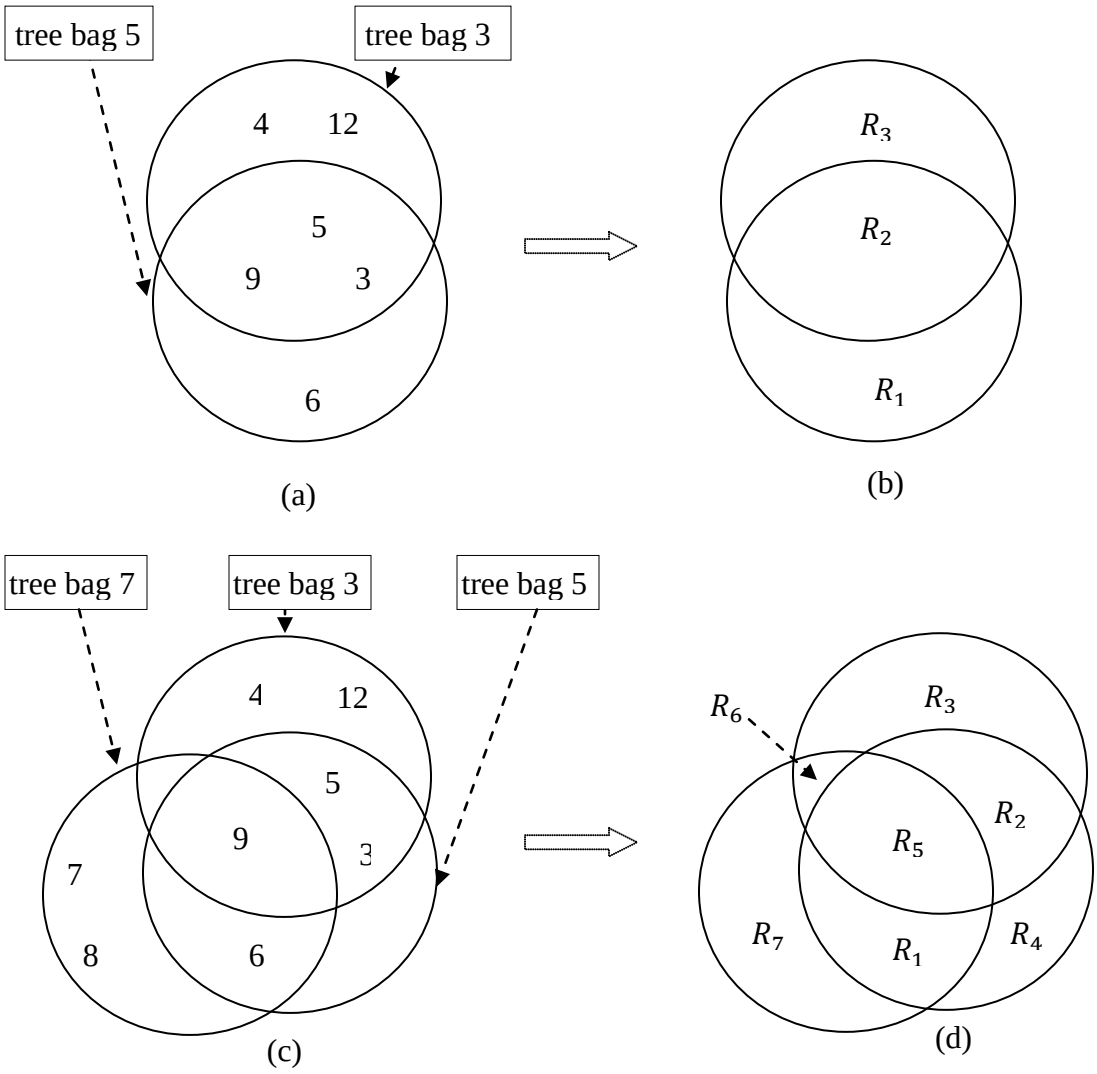


Figure 5.3: Merging spheres. (a) describes the vertices and their designated regions when merging sphere $S_3$ and $S_5$. (b) annotates all the isolated 3 regions in (a) as $R_1$, $R_2$ and $R_3$ separately. For example, vertices {3, 5, 9} in (a) belong to $R_2$ in (b). (c) describes the vertices and their designated regions when adding the sphere $S_7$. (d) annotates all the isolated 7 regions in (c). For example, vertex {3} in (c) is in $R_2$ in (d), but vertices {5, 9} in (c) belong to $R_5$ in (d).

| Vertex | Regions |
|---|---|
| 3 | 2 |
| 4 | 3, 6 |
| 5 | 2 |
| 6 | 1, 4 |
| 7 | 7 |
| 8 | 7 |
| 9 | 5 |
| 12 | 3, 6 |

Table 5.1: The mapping between vertex and its designated space regions

is the designated space for vertices $\{4, 12\}$. Then we still assign $R_6$ in Figure 5.2(d) as the designated space for vertices $\{4, 12\}$ as well. It looks like $R_4$ in Figure 5.2(d) has no associated vertices. For such vertices, we still adopt the same principle. So we assign both $R_1$ and $R_4$ in Figure 5.2(d) as the designated spaces for the vertex $\{6\}$ since $R_4$ in Figure 5.2(d) is part of $R_1$ in Figure 5.2(b), where $R_1$ is assigned for the vertex $\{6\}$. Table 5.1 gives the final mapping between each vertex and its designated space regions. To govern the designated space in a 3D space, we define a function $RF_i$ for the $i$th region.

## 5.2 Candidate Generation

The space partition is serving as a coarse-grained guideline for separating 3D objects conformed to the vertices. The partitioned space tells only what region a vertex belongs to but doesn't tell where exactly the vertex should be positioned. In order to get the fine-grained positioning for each vertex, we need a strategy to generate discrete positions, also called candidates, for vertices.

Before we can generate candidates, we first need a way to divide the designated spaces. One solution is to use a predefined small cubic to divide the designed region of each vertex into many smaller sub-regions. All of the sub-regions which satisfy the region function $RF_i$ will be

valid sub-regions. Then all valid sub-regions serve as the candidate pool for vertices. Figure 5.4 gives an example for the conceptualized view from the plane.

There are two ways to generate candidates for each vertex given the valid cubic pools: arbitrary selection and sampling-based selection. The arbitrary selection is an iterative two step process. The first step arbitrarily generates k candidates for each vertex in its designated space regions. The second step calculates the number of valid combinations based on the constraint F. If the number of valid combinations is less than a predefined threshold, then we repeat the process until the number of valid combinations is equal or greater than the predefined threshold. The importance of the threshold is to guarantee the local optimal since the candidates of each vertex are arbitrary generated. We can extend the arbitrary selection into a biased arbitrary selection by integrating the constraint F into the selection process to decrease the unnecessary repeated computations. The sampling-based selection seeks heuristics insights from known structures. It first runs the space partition on known structures. Then it builds a statistical model to guide more efficient candidate generation.

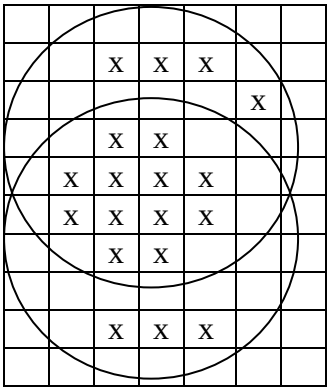

Figure 5.4: Discrete positions of a sphere. The regions marked as “x” are valid.

### 5.3 Dynamic Programming

Given the tree decomposition, the 3D candidates of each vertex, the biological constraints and an adopted fitness scheme, we can search for the optimal folding based on a bottom up dynamic programming. This resembles the TDDP we discussed on chapter 2 and chapter 4.

# CHAPTER 6

# CONCLUSION AND FUTURE WORK

Biopolymer tertiary structure prediction by computer programs is a very important approach to complement the experimental determination of unknown structures of newly identified sequences. Due to the issues of computational complexity and accuracy when considering residue interactions, biopolymer tertiary structure prediction will continue to be an active and challenging research area. In this research, we first discussed two biopolymer tertiary structure prediction methods: template-based and *ab initio* predictions. We then introduced the basic idea of graph tree decomposition and its application to a maximum independent set problem. A small treewidth of biopolymer tertiary structures shows great promise for speeding up the computational complexity by adopting the topology graph model and a suitable TDDP algorithm.

We illustrated how to apply the topology graph model for both template-based and *ab initio* prediction methods. For the template-based prediction method, we developed a heuristic strategy to decrease the space complexity by at least an order of magnitude without sacrificing the speed. We implemented the strategy for protein tertiary structure prediction. At the same time, we also discussed the computation with energy functions in detail. For the *ab initio* prediction method, we proposed a potential topology graph model framework. We identified five basic steps to carry out the prediction: (1) deriving a topology graph and its tree decomposition,

(2) building a sphere for each tree bag, (3) merging spheres, (4) generating candidates and (5) searching the optimal folding by a dynamic programming algorithm.

The throughput and accuracy of the TDDP algorithm we developed for the template-based protein tertiary structure prediction need to be tested from real data in the near future. In addition, we realize the new challenges when extending the topology graph model to *ab initio* method. Partitioning 3D space and generating candidates play a key role for the overall computational complexity and accuracy of the prediction. The soundness of partitioning space is the foundation for the argument that we only need look up the direct children tables when carrying out the TDDP algorithm. The approaches of generating candidates affect the geometric computational complexity and the accuracy of positioning. The framework needs to be continuously refined as testing on known tertiary structures are conducted in the near future.

REFERENCES


[1] Abraham, M., Dror, O., Nussinov, R., and Wolfson, H.J. (2008). Analysis and classification of RNA tertiary structures. *RNA.* 14: 2274-2289.

[2] Auffinger, P., Louise-May, S., and Westhof, E. (1999). Molecular Dynamics Simulations of Solvated yeast tRNA (Asp). *Biophys*. 76: 50-64.

[3] Arnborg, S., and proskurowski, A. (1986). Characterization and recognition of partial 3-trees. *SIAM Journal on Algebraic and Discrete Methods archive*. 7(2): 305-314.

[4] Bodlaender, H.L. (1993). A tourist guide through treewidth. *Technical Report RUU-CS-92-12*, Utrecht University, March 1992, Revised March 1993. 1-14.

[5] Bodlaender, H.L. (1997). Treewidth : Algorithm techniques and results. *Lecture Notes in Computer Science*. 1295: 19 – 36.

[6] Bodlaender, H.L. (2005). Discovering Treewidth. *SOFSEM*. 1-16.

[7] Bodlaender, H.L. (2006). Treewidth : Characterizations, Applications, and Computations. *Technical Report UU-CS-92-12*, Utrecht University. 1-13.

[8] Bowie, J., Luthy, R., and Eisenberg, D. (1991). A Method to Identify Protein Sequences that Fold into a Known Three-Dimensional Structure. *Science*. 253: 164-170

[9] Bryant, S.H., and Altschul, S.F. (1995). Statistics of Sequence-Structure Threading. *Current Opinion Structural Biology*. 5:236-244.

[10] Das, R., and Baker, D. (2007). Automated de novo Prediction of Native-like RNA Tertiary Structure. *Proceedings of the National Academy of Sciences U.S.A.* 104: 14644-14669.

[11] Durbin, R., Eddy, S., Krogh, A., and Mitchison, G. (1998). *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press.

[12] Gherghe, C.M., Leonard, C.W., Ding, F., Dokholyan, N.V., and Weeks, K.M., (2009). Native-like RNA tertiary structures using a sequence-encoded cleavage agent and refinement by discrete molecular dynamics. *Journal of the American Chemical Society*. 131: 2541-2546.

[13] Harvey, S. C., Wang, C., Teletchea, S., and Lavery, R. (2003). Motifs in nucleic acids: molecular mechanics restraints for base pairing and base stacking. *Journal of Computing Chemistry*. 24 (1): 1-9.

[14] Huang, Z., Wu, Y., Robertson, J., Feng, L., Malmberg, R., and Cai, L. (2008). Fast and accurate search for non-coding RNA pseudoknot structures in genomes. *Bioinformatics*. 24:2281-2287.

[15] Jonikas, M.A., Radmer, R.J., Laederach, A., Das, R., Pearlman, S., Herschlag, D., and Altman, R. (2009). Coarse-grained modeling of large RNA molecules with knowledge-based potential and structural filters. *RNA.* 15: 189-199.

[16] Lathrop, R.H., Rogers, R. G., Bienkowska, J., Bryant, B.K.M., Buturovic, L.J., Gaitatzes, C., Nambudripad, R., White, J.V., and Smith, T.F. (1998). Analysis and Algorithms for Protein Sequence-Structure Alignment. *Computational Method in Molecular Biology*. Salzberg, S.L., Searls, D.B., and Kasif, S., (eds). Elsevier. 227-255

[17] Lenhof, H. P., Reinert, K., and Vingron, M. (1998). A Polyhedral Approach to RNA Sequence Structure Alignment. *Journal of Computational Biology*. 5(3): 517-530

[18] Li, H. (2007). Computing images of protein cores for protein threading (MS Thesis). The University of Georgia.

[19] Major, F., Gautheret, D., and Cedergren, R., (1993). Reproducing the three-dimensional structure of a tRNA molecule from structural constraints, *Proceedings of National Academy Science*. USA. 90: 9408-9412.

[20] Major, F., Turcotte, M., Gautheret, D., Lapalme, G., Fillion, E., and Cedergren, R. (1991). The combination of symbolic and numerical computation for three-dimensional modeling of RNA, *Science*, 253(5025): 1255-1260.

[21] Malhotra, A., Tan, R., and Harvey, S. (1994). Modeling large RNAs and ribonucleoprotein parti-cles using molecular mechanics techniques. *Biophysical Journal*. 66 (6): 1777-1795.

[22] Malmberg, R. and Cai, L. (2010). From topology to 3D: effective RNA tertiary structure prediction. *Lab Report*, University of Georgia

[23] Kim, D., Xu, D., Guo, J., Ellrott, K. and Xu, Y. (2003). PROSPECT II: protein structure prediction program for genome-scale applications. Protein Engineering. 16(9):641-650.

[24] Parisien, M. and Major, F., (2008). The MC-Fold and MC-Sym pipeline infers RNA structures from sequence data. *Nature*. 452: 51-55.

[25] Robertson, N., and Seymour, P.D. (1984). Graph minors III: Planar tree-width. *Journal of Combinatorial Theory*, Series B 36: 49–64,

[26] Sharma, S., Ding, F., and Dokholyan, N.V. (2008). iFoldRNA: three-dimensional RNA structure prediction and folding. *Bioinformatics.* 24 (17) : 1951-1952.

[27] Smith, T. F., Conte, L.L., Bienkowska, J., Gaitatzes, C., Rogers, R., and Lathro, R. (1997). Current limitations to protein threading approaches. *Journal of Computational Biology*. 4(3): 217-225.

[28] Song, Y., Ellrott, K., Liu, C., Guo, J., Xu,Y., and Cai, L. (2005). "Efficient Protein Threading with Tree Decomposition" manuscript. The University of Georgia

[29] Song, Y., Liu, C., Huang, X., Malmberg, R., Xu,Y., and Cai, L. (2006). Efficient Parameterized Algorithms for Biopolymer Structure-Sequence Alignment. *IEEE/ACM Transactions of on Computational Biology and Bioinformatics.* 3(4): 423-432.

[30] Wang, Y., Mohebbi, M., Lin, D., Malmberg, R., and Cai, L. (2010). Tree width of RNA tertiary structures in PDB and NDB, *Lab Report*, University of Georgia.

[31] Westhof, E., and Auffinger, P. (2000). RNA Tertiary Structure. John Wiley & Sons Ltd. Chichester.

[32] Xu, J. (2005). Rapid Side-Chain Packing via Tree Decomposition. *Proceeding of 2005 International Conference Research in Computational Biology*. 423-439.

[33] Xu, J., Li, M., Kim, D., and Xu, Y. (2003). RAPTOR : Optimal Protein Threading by Linear Programming. *Journal of Bioinformatics and Computational Biology*. 1(1): 95-113.

[34] Xu, J., Jiao, F., and Berger, B. (2005). A Tree-Decomposition Approach to Protein Structure Prediction. *Proceeding of 2005 IEEE Computational Systems Bioinformatics Conference.* 247-256.

[35] Xu,Y., Liu, Z., Cai, L., and Xu, D. (2007). Protein Structure Prediction by Protein Threading. *Computational Methods for Protein Structure Prediction and Modeling: Volume 2: Structure Prediction.* Xu,Y., Xu, D., and Liang J. (eds). Springer

[36] Xu, Y., Xu, D., and Uberbacher, E.C. (1998). An Efficient Computational Method for Globally Optimal Threading. *Journal of Computational Biology*. 5(3): 597-614.