Brandeis University

Senior Thesis in Computer Science

Graph Matching, Pattern Learning, and Protein Modeling

Wei Qian

Supervised by
Prof. Pengyu Hong

May 5, 2017

Contents

1 Graph Matching **5**
- 1.1 Introduction . . . 5
- 1.2 Syntax and Definition . . . 5
- 1.3 Problem Definition . . . 6
- 1.4 Graph Matching by Gold and Rangarajan . . . 6
- 1.5 Implementation Detail . . . 8
  - 1.5.1 Compatibility . . . 8
  - 1.5.2 Parallel Computing and Caching . . . 8
  - 1.5.3 Heuristic . . . 8
  - 1.5.4 Converge Condition . . . 8
  - 1.5.5 Run Time . . . 8
- 1.6 Testing the Algorithm . . . 9
- 1.7 Improvement and Modification . . . 9
  - 1.7.1 Local Minima . . . 9
  - 1.7.2 High Recall + Low Precision . . . 11
- 1.8 Conclusion . . . 12

2 Pattern Learning **13**
- 2.1 Introduction . . . 13
- 2.2 Syntax and Definition . . . 14
- 2.3 Problem Definition . . . 15
- 2.4 Initializing the Components . . . 15
- 2.5 EM Algorithm . . . 16
  - 2.5.1 Estimation Step . . . 16
  - 2.5.2 Maximization Step . . . 16
- 2.6 Detect the Spatial Pattern . . . 19
- 2.7 Implementation Detail . . . 19
  - 2.7.1 Coding Complexity . . . 19
  - 2.7.2 Null Node in the Model . . . 19
  - 2.7.3 Run Time . . . 19
- 2.8 Testing the Model . . . 20
- 2.9 Conclusion . . . 20

3 Protein Modeling **21**
- 3.1 Introduction . . . 21
- 3.2 Protein to ARG . . . 22
  - 3.2.1 Edge for Protein ARG . . . 22
  - 3.2.2 Node for Protein ARG . . . 23
- 3.3 Protein Backbone Encoding in Objective Function . . . 26
- 3.4 Proof of Concept . . . 26
- 3.5 Conclusion . . . 27

4 Looking Forward **28**
4.1 Graph Matching . . . 28
4.1.1 More Efficient Implementation . . . 28
4.1.2 Automate Parameters Tuning . . . 29
4.2 Pattern Learning . . . 29
4.2.1 Smarter Component Initiation . . . 29
4.2.2 Component and Node Deletion . . . 29
4.2.3 Other Applications . . . 30
4.3 Protein Modeling . . . 31
4.3.1 Novel Structure Motif in Protein . . . 31
4.3.2 Protein as Documents and Amino Acid as Word . . . 31
4.3.3 Edge for Protein ARG Revisit . . . 31

Abstract

One of the most amazing capabilities of human beings is to extract common spatial patterns from observations and use these patterns to make inferences - recognizing a car by summarizing the important components of cars, including rims, windows, and trunks etc., and their spatial relationship while ignoring the specific design of different cars.

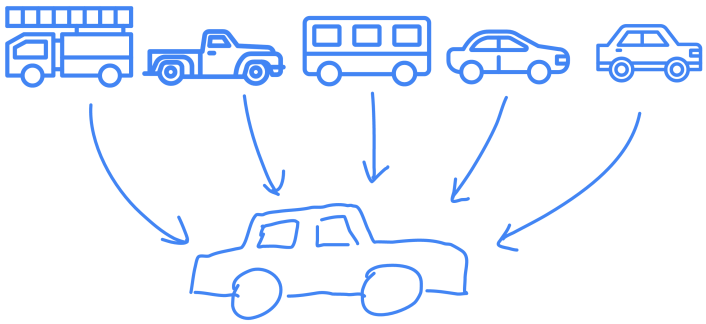

Figure 1: *Different cars can be summarized as some simple components in a certain spatial relationship.*[1]

Here, we turn a set of spatial representations into a set of attributed relational graphs (ARGs), which consist of nodes and edges with a vector representing their individual features, and train a probabilistic parametric model that summarize the common sub-graph of the ARGs set (i.e. the common spatial pattern from the set of spatial representations).

Our key contributions are 1) introducing a stochastic process to the graph matching step which utilizes a graduated assignment algorithm, 2) adding a null node network in each ARG to avoid matches among background nodes (i.e. node not in the pattern), and applying the model to crystallography protein structure data to learn the common structure among proteins share a certain function. To apply the general algorithm to our protein data and model the structure data as ARG, we 3) introduce an additional term in our objective function representing the protein backbone, and 4) use a local substitution vector to model the similarity between two different amino acids based on their specific local environment.

Graph is a powerful representation and can be used to model a lot of things like neuron morphologies and social network. Utilizing such algorithm can help us and machines to understand many of the patterns in real life.

[1]The figure is drawn by a tool called Autodraw developed by Google using machine learning techniques.

Acknowledgments

This work was supported by the Jerome A. Schiff Fellowship, and advised by Prof. Pengyu Hong at Brandeis University.

Prof. Hong gave me a shot in the summer of 2015, and asked me to implement some pre-existing algorithms in MATLAB. We then modified this algorithm to handle some issues that occurred with larger graphs. After we fixed the issues, we applied the modified algorithm to protein structure data resulting in this work. I thank Prof. Hong for his continuous support, inspiration, and guidance along this amazing journey.

I came to Brandeis University as a math major, but the magical intro lecture by Prof. Antonella DiLillo led me to the computer science major instead. The JBS program led by Prof. Timothy J. Hickey and Prof. Marie M. Meteer allowed me to build an Siri-like app and introduced me to this amazing AI world. Prof. Harry Mairson introduced the beauty of simplicity to me, while courses taught by Prof. Mitch Cherniack and Prof. Liuba Shrira helped me appreciate such simplicity in even the most powerful and complex systems. Such "deconstruction" mindset guided me through many obstacles along this research journey. There are many other mentors at Brandeis helped me appreciate what I do and inspired me to do great work in the future, so I also want to give a special shout out to this warm welcoming community.

Last but not least, I want to thank my family, friends, and Netflix for distracting me from research and being the scapegoats for my own procrastination.

Chapter 1

Graph Matching

1.1 Introduction

In order to extract the common pattern from graphs, we need to first compare two graphs and match[1]the relevant part together to better summarize them. However, since the largest common subgraph problem is NP-complete, the graph matching problem we are dealing with is also NP-complete (even though you can argue it's NP-hard in some other definition). Therefore, we must look for good suboptimal solutions via approximation.

There are two main approaches for graph matching. One approach involves the construction of a state-space that can be searched via some brach and bound methods. While some heuristic rule can help us reduce the complexity from exponential to polynomial, the polynomial complexity would still have a high complexity. The second approach employs nonlinear optimization methods like relaxation labeling, which do not search based on the state-space and generally have a much lower computational complexity.

In the realm of nonlinear optimization methods and similar to relaxation labeling, here we described a graduated assigned approach developed by Gold and Rangarajan in 1996 and the improvement we made on the algorithm in order deal with larger graphs we encounter today 20 years later.

1.2 Syntax and Definition

Here we use a attributed relational graph (ARG) to represent our spatial pattern. An ARG, G, is a *directional* graph with labels on its nodes ($\overrightarrow{N_a}$ is the label for the ath node in the graph) and edges ($\overrightarrow{E_{ab}}$ is the label for the edge from the ath node to the bth node).

Since two graphs might only match partially by their subgraph or might not have match at all, Gold and Rangarajan add a null node, ϕ, in each ARG.

We then can define the node compatibility function(c_N) and the edge compatibility function(c_E) to calculate the compatibility, C, between the nodes and edges between two ARGs, G and G'. Here, C_{ai} is the compatibility between node $\overrightarrow{N_a}$ in G and node $\overrightarrow{N_i}$ in G', while C_{abij} is the compatibility between edge $\overrightarrow{E_{ab}}$ in G and $\overrightarrow{E_{ij}}$ in G'.

[1]The graph matching here is different than the "matching" problem in graph theory where the *matching* is a set of pairwise non-adjacent edge.

The match result is defined as a matrix M where M_{ai} is the probability matching the ath node in G to the ith node in G'.

1.3 Problem Definition

We defined the problem of ARG matching in the following manner. Given two ARG, G and G', we want to find the match matrix M such that the following objective function is minimized:

$$E(M) = -\frac{1}{2}\sum_{a=1}^{G}\sum_{i=1}^{G'}\sum_{b=1}^{G}\sum_{j=1}^{G'} M_{ai}M_{bj}C_{abij} - \alpha\sum_{a=1}^{G}\sum_{i=1}^{G'} M_{ai}C_{ai} \tag{1.1}$$

subject to a two way constraints: $\forall a \neq \phi$, $\sum_{i=1}^{G'} M_{ai} = 1$ and $\forall i \neq \phi$, $\sum_{i=1}^{G} M_{ai} = 1$:

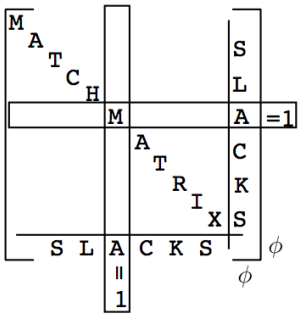

Figure 1.1: *Both rows and columns should sum up to 1 except the row and column involving ϕ nodes, which are considered the slack row and slack column.*

1.4 Graph Matching by Gold and Rangarajan

In Gold and Rangarajan's original algorithm, they solve the problem using a graduated assignment method where the two-way constraints are satisfied by normalizing rows and columns iteratively for many rounds (Sinkhorn 1963) and their compatibilities are defined as:

$$C_{ai} = \begin{cases} 0 & a = \phi \bigcup i = \phi \\ c_N(\overrightarrow{N_a}, \overrightarrow{N_i}) & otherwise \end{cases} \tag{1.2}$$

$$C_{abij} = \begin{cases} 0 & a = \phi \bigcup b = \phi \bigcup i = \phi \bigcup j = \phi \\ c_E(\overrightarrow{E_{ab}}, \overrightarrow{E_{ij}}) & otherwise \end{cases} \tag{1.3}$$

To expand on their graduated assignment method, we will first need to convert our objective function to an assignment problem. They way we do it is through Taylor series approximation, where given an initial M^0, we would have:

$$\begin{aligned} E(M) &= -\frac{1}{2}\sum_{a=1}^{G}\sum_{i=1}^{G'}\sum_{b=1}^{G}\sum_{j=1}^{G'} M_{ai}M_{bj}C_{abij} - \alpha\sum_{a=1}^{G}\sum_{i=1}^{G'} M_{ai}C_{ai} \\ &= -\frac{1}{2}\sum_{a=1}^{G}\sum_{i=1}^{G'}\sum_{b=1}^{G}\sum_{j=1}^{G'} M^0_{ai}M^0_{bj}C_{abij} - \alpha\sum_{a=1}^{G}\sum_{i=1}^{G'} M^0_{ai}C_{ai} - Q_{ai}(M_{ai} - M^0_{ai}) \end{aligned} \tag{1.4}$$

where

$$Q_{ai} = -\frac{\partial E}{\partial M_{ai}}\bigg|_{M=M_0} = +\sum_{b=1}^{G}\sum_{j=1}^{G'} M_{bj}^0 C_{abij} + \alpha C_{ai} \tag{1.5}$$

Therefore, minimizing our objective function E based on our Taylor series expansion is equivalent to maximizing

$$+\sum_{a=1}^{G}\sum_{i=1}^{G'} Q_{ai} M_{ai} \tag{1.6}$$

which is an assignment problem!

Therefore, we can run the following algorithm for the graduated assignment:

Initialize β to β_0, M_{ai} to a random sample from $U(0,1)$
Begin A**:** (Do A until $\beta \geq \beta_f$)
  Begin B**:** (Do B until M converges or # of iterations $> I_0$)
  $Q_{ai} \leftarrow \sum_{b=1}^{G}\sum_{j=1}^{G'} M_{bj}^0 C_{abij} + \alpha C_{ai}$
  $M_{ai}^0 \leftarrow exp(\beta Q_{qi})$
    Begin C**:** (Do C until M converges or # of iterations $> I_1$)
    Update M by normalizing across all rows:
    $M_{ai}^1 = \frac{M_{ai}^0}{\sum_{i=1}^{G'} M_{ai}^0}$ for all $a \neq \phi$
    Update M by normalizing across all columns:
    $M_{ai}^0 = \frac{M_{ai}^1}{\sum_{a=1}^{G} M_{ai}^1}$ for all $i \neq \phi$
    End C
  End B
$\beta \leftarrow \beta_r \beta$
End A
Perform Clean-up Heuristic

Variable and constant definitions can be found as:

β	control parameter of the continuation method
β_0	initial value of the control parameter β
β_f	maximum value of the control parameter β
β_r	rate at which the control parameter β is increased
E_{wg}	graph matching objective, equation (1)
$\{M_{ai}\}$	match matrix variables
$\{\hat{M}_{ai}\}$	match matrix variables including the slacks (see Figure 2)
$\{Q_{ai}\}$	partial derivative of E_{wg} with respect to M_{ai}
I_0	maximum # of iterations allowed at each value of the control parameter, β
I_1	maximum # of iterations allowed for Sinkhorn's method
	(back and forth row and column normalizations)

1.5 Implementation Detail

1.5.1 Compatibility

To calculate the compatibility, we assume the label/feature follows a Gaussian probability distribution function so that the compatibility function c_N and c_G is defined as:

$$c_N(\overrightarrow{a}, \overrightarrow{b}) = c_E(\overrightarrow{a}, \overrightarrow{b}) = \frac{exp(-\frac{1}{2}(\overrightarrow{a} - \overrightarrow{b})^T \Sigma^{-1}(\overrightarrow{a} - \overrightarrow{b})}{(2\pi)^{\zeta/2}|\Sigma|^{1/2}} \tag{1.7}$$

where δ is the covariance matrix and ζ is the dimension of $\overrightarrow{a}$ and $\overrightarrow{b}$. Since we assume feature independence, the covariance matrix δ is the identity matrix, which essentially will give us:

$$c_N(\overrightarrow{a}, \overrightarrow{b}) = c_E(\overrightarrow{a}, \overrightarrow{b}) = \frac{exp(-\frac{1}{2}(\overrightarrow{a} - \overrightarrow{b})^T (\overrightarrow{a} - \overrightarrow{b})}{(2\pi)^{\zeta/2}} \tag{1.8}$$

1.5.2 Parallel Computing and Caching

The majority of the computation complexity in the algorithm is in computing the compatibility, especially the edge compatibility.

Since the compatibility function defined can be computer independently, one way we could speed up the algorithm is calculating the compatibility in parallel fashion.

Another way we could improve the computation time would be through caching, where we can pre compute a node and edge compatibility matrix, as the label is never updated. However, the edge compatibility matrix can be very large and takes a lot of memory to cache, so you might also want to use data structure like *sparse matrix* since the connection matrix (representing edge) is likely to be sparse as well.

1.5.3 Heuristic

In our implementation, we use a very simple heuristic to clean up the match matrix M from a *softassign* matrix to a matrix only with 0 and 1. We simply set the largest value (at index j) for each row i to 1 and other value to 0. After setting M_{ij} to 1, we also set the entire j column to 0 in order to satisfy the two-way constraints.

If you want to be more fancy about this, you can also convert this heuristic problem to a maximum spanning tree problem to make sure all the M_{ij} that you set to 1 have the largest sum.

1.5.4 Converge Condition

In our implementation, we compare the previous match matrix $M^{(0)}$ and the current match matrix $M^{(1)}$, and conclude the matrix converge if the difference is less than some threshold ι:

$$\sum_{a=1}^{G}\sum_{i=1}^{G'} |M_{ai}^{(0)} - M_{ai}^{(1)}| < \iota \tag{1.9}$$

1.5.5 Run Time

The algorithm takes $\sim 15s$ to complete when matching two graphs with ~ 30 nodes. However, you can certainly adjust the parameter like I and β to achieve faster run time.

1.6 Testing the Algorithm

To test the algorithm, we first set a noise level $\epsilon \in [0, 1]$. Then we generate a pattern with n nodes and embed this pattern to 2 randomly generated graphs, G and G', with n to $n * (1 + \epsilon)$ nodes. Once we embed the pattern in these two random graphs, we shuffle the index for each node and add some noise to the node and edge labels:

$$\overrightarrow{N_a} \leftarrow \overrightarrow{N_a} + \chi * \overline{N} \text{ where } \chi \in [0, \epsilon] \tag{1.10}$$

$$\overrightarrow{E_{ab}} \leftarrow \overrightarrow{E_{ab}} + \chi * \overline{E} \text{ where } \chi \in [0, \epsilon] \tag{1.11}$$

We then run the matching algorithm $match(G, G')$ and get a match matrix M. We ignore the match with the null node ϕ, and calculate the total number of correct match m, and the total number of match observed l from the match matrix M. Then we can evaluate the performance by $precision$, $recall$ and $F_1 score$:

$$precision = \frac{m}{l} \tag{1.12}$$

$$recall = \frac{m}{n} \tag{1.13}$$

$$F_1 = 2 * \frac{precision * recall}{precision + recall} \tag{1.14}$$

1.7 Improvement and Modification

While the original algorithm works well with small graphs with 20 to 30 nodes, we run into some problems while matching larger graphs with 50 to 100 nodes and therefore make some improvement to Gold and Rangarajan's original algorithm.

1.7.1 Local Minima

The first problem we encounter is local minima. In this case, the algorithm will generate correct and even perfect match for majority of the test cases. However, for some of the test case, the match result is entirely wrong, which indicates that the graduated assignment process stuck in some local minima.

In the original algorithm, if we initialized M_0 in the wrong places, the assignment process can easily get stuck in sub-optimal match (local minima). While adjusting β and β_r can mediate the problem, it does not work well in larger graphs since there are more sub-optimal match. Therefore, we introduce a stochastic process in the algorithm:

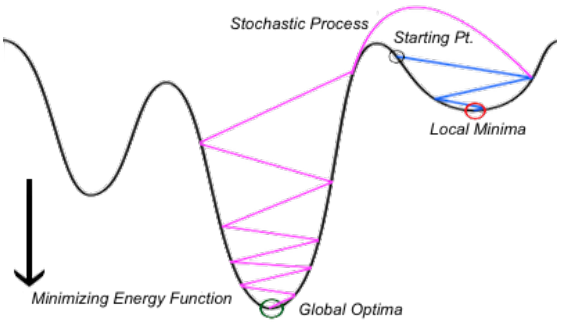

Figure 1.2: *The intuition behind introducing the stochastic process.*

In the stochastic process, we essentially add some small noise to the match matrix at the beginning of step B:

$$M_{ai}^0 \leftarrow M_{ai}^0 + \frac{\tau * U(-1,1)}{|G|} \tag{1.15}$$

where τ is the stochastic/noise level, $|G|$ is the number of node in G, and $U(-1,1)$ is a random sample from a uniform distribution between -1 and $+1$.

Therefore, the updated algorithm becomes:

Initialize β to β_0, M_{ai} to a random sample from $U(0,1)$
Begin A**:** (Do A until $\beta \geq \beta_f$)
  Begin B**:** (Do B until M converges or # of iterations $> I_0$)
  $M_{ai}^0 \leftarrow M_{ai}^0 + \frac{\tau * U(-1,1)}{|G|}$
  $Q_{ai} \leftarrow \sum_{b=1}^{G} \sum_{j=1}^{G'} M_{bj}^0 C_{abij} + \alpha C_{ai}$
  $M_{ai}^0 \leftarrow exp(\beta Q_{qi})$
    Begin C**:** (Do C until M converges or # of iterations $> I_1$)
    Update M by normalizing across all rows:
    $M_{ai}^1 = \frac{M_{ai}^0}{\sum_{i=1}^{G'} M_{ai}^0}$ for all $a \neq \phi$
    Update M by normalizing across all columns:
    $M_{ai}^0 = \frac{M_{ai}^1}{\sum_{a=1}^{G} M_{ai}^1}$ for all $i \neq \phi$
    End C
  End B
$\beta \leftarrow \beta_r \beta$
End A
Perform Clean-up Heuristic

Once we incorporate the stochastic process to the algorithm, we gain a significant improvement in F_1 score and there is very little chance the match result is entirely wrong:

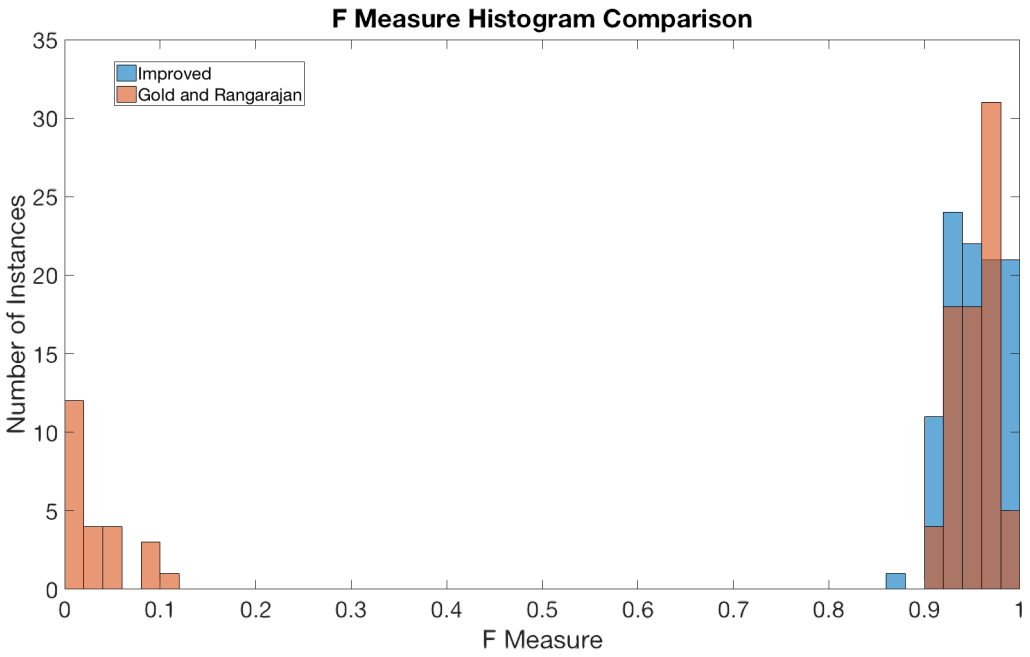

Figure 1.3: F_1 *score distribution histogram for the original algorithm (orange) and the improved algorithm with stochastic process (blue).*

1.7.2 High Recall + Low Precision

The other challenge we encounter is that even though we get high recall rate in a test, we also get low precision rate, which means the algorithm can generate correct match but also force many of the background nodes (i.e. nodes outside of the pattern) matching to each other.

The first explanation came across our mind would be the two test graphs we generated can share pattern outside the embedded one. However, after careful examination, this is proven not the case.

Therefore, we examed how Gold and Rangarajan's original algorithm matches background nodes in G to the null node ϕ in G'. We realized that since the original algorithm define the compatibility function so that any compatibility related to the ϕ node will be set to 0. Therefore, the algorithm does not *actively* match background nodes to ϕ node, but *hope* that the background noise does not attract to one specific node, and end up matching with the ϕ node.

To handle this problem gracefully, we essentially want to create a fully connected null node network that can match to any of the background pattern. Therefore, we rewrite the compatibility function as:

$$C_{ai} = \begin{cases} 0 & a, i = \phi \\ c_N(\overrightarrow{N_a}, \overrightarrow{N_i}) & a, i \neq \phi \\ p \text{ percentile of } [C_{1i}, C_{2i}, ...C_{|G|-1i}] & a = \phi \\ p \text{ percentile of } [C_{a1}, C_{a2}, ...C_{a|G'|-1}] & i = \phi \end{cases} \tag{1.16}$$

$$C_{abij} = \begin{cases} c_E(\overrightarrow{E_{ab}}, \overrightarrow{E_{ij}}) & a, b, i, j \neq \phi \\ p \text{ percentile of } \{C_{abij} | a, b, i, j \neq \phi\} & a, b \neq \phi \bigcap i, j = \phi \\ p \text{ percentile of } \{C_{abij} | a, b, i, j \neq \phi\} & a, b = \phi \bigcap i, j \neq \phi \\ 0 & otherwise \end{cases} \tag{1.17}$$

where p is a percentile we can adjust so that set to 100 to give null node/edge the maximum compatibility every seen while set to 0 to give the minimum compatibility. You can adjust the p to balance the recall and precision rate since higher percentile will encourage nodes match to null node result in low recall but high precision while lower percentile will allow background nodes match together result in high recall but low precision.

While this is a lot to unpack, the updated compatibility function essentially create a self-linked edge for null node $\overrightarrow{E_{\phi\phi}}$ and give definition for the compatibility of null node to other real node and self-linked null edge to other edge linking two real nodes. Since we don't normalized the slack row and column (match result for null node), this is equivalent to create a fully connected null node network:

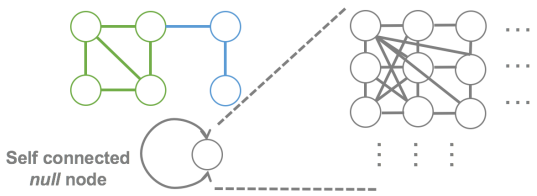

Figure 1.4: *The self linked/connected null node ϕ (grey) can be treated as a fully connected null node that can be matched to any pattern including background nodes.*

After modified the compatibility function in the original algorithm, we are able to generate match result with both high recall and high precision. For instance, two graph with backgrounds nodes at the beginning and the end (in terms of their index) are not forced to match to each other (left) and end up match to the null node/the slack column/row in the end (right):

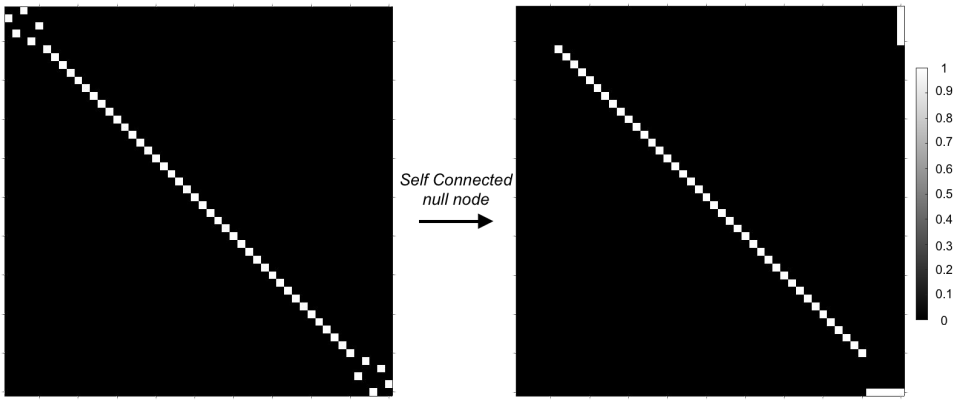

Figure 1.5: *This is a visualization of the match matrixes produced by the original algorithm by Gold and Rangarajan (left) and the modified version (right).*

1.8 Conclusion

In this chapter we introduced a graph matching algorithm that we improved and used to match the common sub-graph of two ARGs. If we do not apply the heuristic function, the match matrix M shows us how likely one node in graph G can be matched to another graph G', which can in turn help us the summarize the common pattern among a set of graphs.

Chapter 2

Pattern Learning

2.1 Introduction

Here in pattern learning, we extract the common pattern from a set of ARGs, and the extracted information can then be used to summarize the given ARGs and predict if a new ARG contains the common pattern we summarized. In our work, for instance, we can extract the common structure of a set of proteins that share the same function. The common structure we extracted can give us insight on how this structure can perform such function, and if a new protein also has the same structure that can carry out the same function.

To perform pattern learning, we utilize a probabilistic parametric model to represent the common pattern from the graphs (Hong[1]and Huang 2004). Similar to how three normal distribution (or components) can capture the data distribution generated by $f(x)$ below, we used a couple component ARGs with various mean and variance to represent the common pattern:

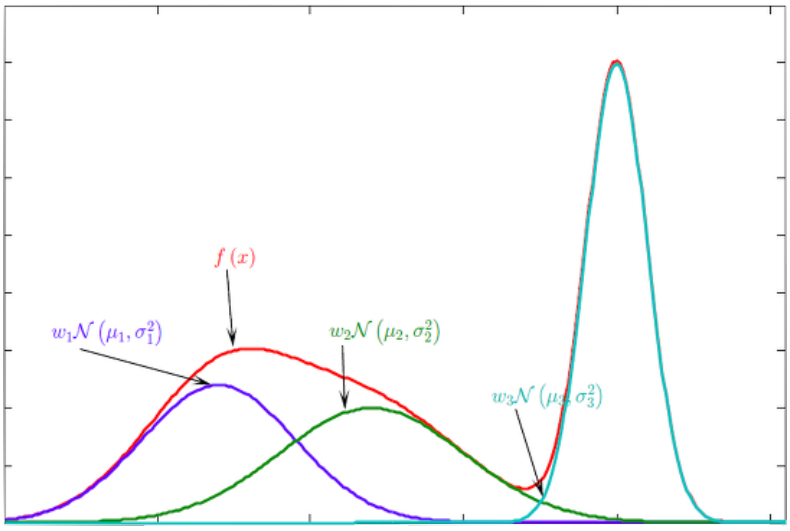

Figure 2.1: *The data distribution generated by $f(x)$ can be captured by three normal distribution with various means and variances.*

Therefore, the model training process is essentially training and setting up some component ARGs so that all the nodes and edges have the means and variances that best capture the common pattern in the given set of ARGs.

[1]My dear advisor!

Based on the algorithm described by Hong and Huang, we first pick some ARGs from the input sample ARGs, and initialize them as the component ARGs. Then we perform an EM algorithm where on the **E**xpectation step we run the graph matching algorithm to calculate the matching probability between model ARGs and sample ARGs while on the **M**aximization step we update the model ARG (e.g. updating means, variances, and deleting redundant nodes) in order to achieve higher matching probability. Finally, once the update is very small, we output the model ARGs to represent the common pattern in sample ARGs:

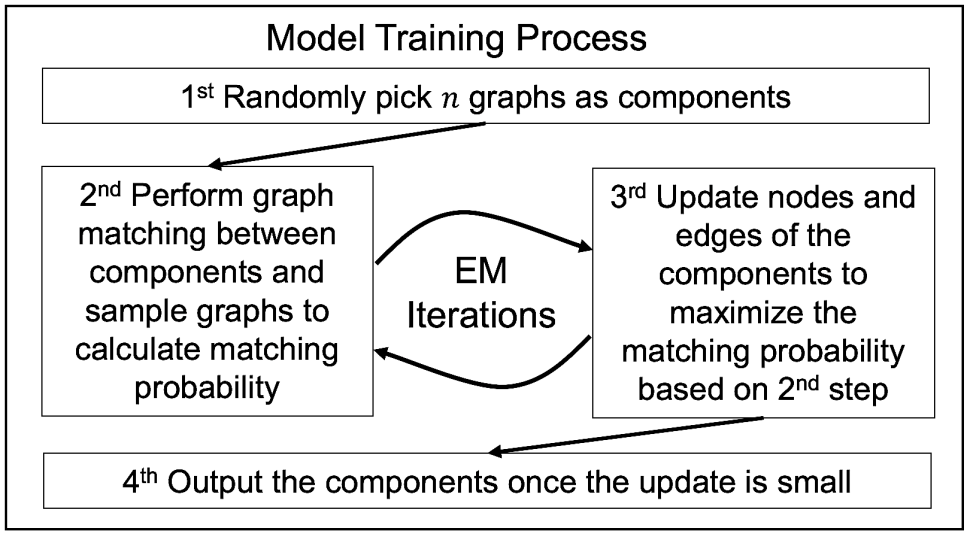

Figure 2.2: *The probabilistic parametric model training process with EM algorithm.*

2.2 Syntax and Definition

The sample ARGs (i.e. the set of ARGs we want to learn/extract pattern from) is denoted as $\{G_s\}_{s=1}^{S}$ where S is the number of model components. Within each sample ARG, we indicate the label for each node and edge as $\overrightarrow{N_a^s}$ and $\overrightarrow{E_{ab}^s}$. This is similar to what we have in Chapter 1 but with an additional index s indicating which sample G these nodes and edges belong to. In addition, we denoted the real nodes in sample ARG, $G_s - \phi$, as $\widehat{G_s}$.

For the model, we denoted it as Z and the model consists of a set of parametric model components $\{\Phi_w\}_{w=1}^{W}$ where W is the number of model components. For each component, there are also an associated weight α_w which indicates how much information the component captures and how important the component is.

Within each component ARG, we indicate the mean for each node and edge as $\overrightarrow{N_a^w}$ and $\overrightarrow{E_{ab}^w}$ similar to what we have for sample ARGs, but with w instead of s indicating which model this node and edge belongs to. In addition to the mean, there are also the covariance matrixes for node and edge denoted by Σ_a^w and Σ_{ab}^w with a similar index. Last but not least, for each node in each component, there is an associated frequency/weight β_a^w indicating how important one node is and if we can delete such node.

In addition, the matching algorithm introduced in Chapter 1 can generate match matrix M^{sw} between sample ARG G_s and component ARG Φ_w as well as the associated node/edge compatibility C^{sw}.

2.3 Problem Definition

Given a set of sample ARG $\{G_s\}_{s=1}^{S}$, our problem will be inferring the parameters in Z including the number of components W, weight for each component α, mean for each node/edge $\overrightarrow{N^w}/\overrightarrow{E^w}$, and the covariance associated with them Σ.

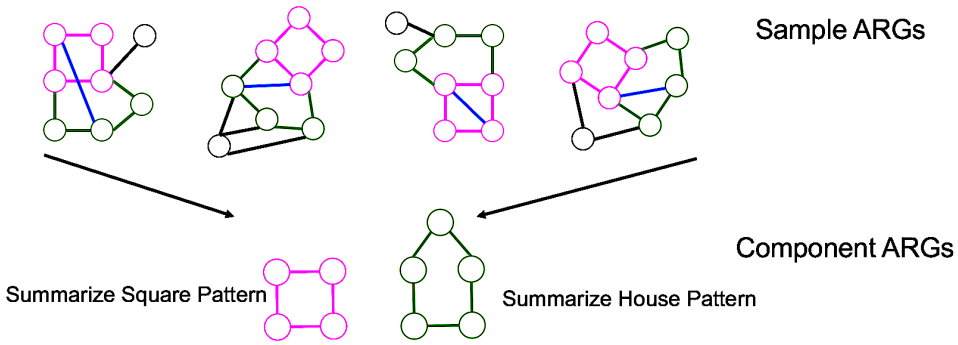

Figure 2.3: *Component can summarize different part of the common pattern.*

2.4 Initializing the Components

The first step of training the model is initializing the components in the model.

The way we do is manually pick W random ARG from our sample set, and initialize them as component ARG Φ. We took their original label as the mean, set their covariance matrix as the identity matrix I, and give equal weight to all the components and nodes. If we are converting a sample G_s to a component Φ_w, we would have:

$$\overrightarrow{N_a^w} \leftarrow \overrightarrow{N_a^s} \qquad \forall a \in \widehat{G_s} \tag{2.1}$$

$$\overrightarrow{E_{ab}^w} \leftarrow \overrightarrow{E_{ab}^s} \qquad \forall a, b \in \widehat{G_s} \tag{2.2}$$

$$\Sigma_a^w \leftarrow I \tag{2.3}$$

$$\Sigma_b^w \leftarrow I \tag{2.4}$$

$$\alpha_w \leftarrow \frac{1}{W} \tag{2.5}$$

$$\beta_a^w \leftarrow \frac{1}{|G_s|} \qquad \forall a \in G_s) \tag{2.6}$$

While we pick W manually based on the complexity of the pattern here, theoretically, we can also start with a relatively larger number of components are delete some of them later. In terms of picking the sample, you can also do better than random by doing a pairwise graph matching among the sample ARG and choose ones that are the most representative based on the match matrixes M.

2.5 EM Algorithm

Once we initialized the components $\{\Phi_w\}_{w=1}^{W}$ with all the parameters, we will then run through an EM algorithm to tune this parameters so our model Z can capture the sample $\{G_s\}_{s=1}^{S}$ at its best. EM algorithm consist of an estimation step where we estimate how well our parameters perform and a maximization step where we tune the parameters to maximize the estimation. We run through these two steps iteratively until the tuning on the parameter is very small.

2.5.1 Estimation Step

In this step we estimate how well model Z representing the sample ARG G as $P(G|Z)$. Once we finish training the model, $f(G_{new}|Z)$ can also be used to predict if a new ARG G_{new} contains the learned/extracted pattern.

The first thing we do in the estimation step is matching each component Φ_w with each sample G_s, and get a match matrix M^{sw} from the graph matching algorithm introduced in Chapter 1. In addition, the algorithm will also return the node and edge compatibility C^{sw} computed as described in Section 1.5.1.

Once we have the match matrix, we can calculate the probability of G_s matching Φ_w:

$$\xi(G_s|\Phi_w) = \sum_{a=1}^{\widehat{G_s}} \sum_{i=1}^{\Phi_w} M_{ai}^{sw} C_{ai}^{sw} + \sum_{a=1}^{\widehat{G_s}} \sum_{b=1}^{\widehat{G_s}} \sum_{i=1}^{\Phi_w} \sum_{j=1}^{\Phi_w} M_{ai}^{sw} M_{bj}^{sw} C_{abij}^{sw} \tag{2.7}$$

$$P(G_s = \Phi_w) = \frac{\xi(G_s|\Phi_w)}{\sum_{t=1}^{W} \xi(G_s|\Phi_t)} \tag{2.8}$$

With $\xi(G_s|\Phi_w)$, we then can calculate:

$$f(G|Z) = \sum_{w=1}^{W} \alpha_w \xi(G|\Phi_w) \tag{2.9}$$

2.5.2 Maximization Step

In the maximization step, we adjust and tune the variable based on $P(G_s = \Phi_w)$, M^{sw} and C^{sw} that we calculated in the Estimation step.

Update Component Weight

First we update the component weight α_w for each component Φ_w as:

$$\alpha_w = \frac{\sum_{s=1}^{S} P(G_s = \Phi_w)}{S} \tag{2.10}$$

The component weight here is essentially calculating the average probability it matched to all the sample ARGs $\{G_s\}_{s=1}^{S}$.

Update Component Node Frequency

Then we update the frequency (i.e. weight) for each node in each component Φ_w:

$$\beta_a^w = \frac{\sum_{s=1}^{S}\sum_{i=1}^{\widehat{G_s}} M_{ia}^{sw} P(G_s = \Phi_w)}{\sum_{s=1}^{S} |\widehat{G_s}| P(G_s = \Phi_w)} \tag{2.11}$$

The component node frequency is essentially an weighted average of the matching probability of this specific component node in all the matching matrix M^{sw}.

Update Mean for Component Node

To tune the mean for each component node, $\overrightarrow{N_a^w}$, we follow:

$$\overrightarrow{N_a^w} = \frac{\sum_{s=1}^{S}\sum_{i=1}^{\widehat{G_s}} \overrightarrow{N_i^s} M_{ia}^{sw} P(G_s = \Phi_w)}{\sum_{s=1}^{S}\sum_{i=1}^{\widehat{G_s}} M_{ia}^{sw} P(G_s = \Phi_w)} \tag{2.12}$$

Here we essentially calculated a weighted average of all the sample nodes, $\overrightarrow{N_i^s}$, based on how likely our component node matches to that sample node in match matrix M^{sw} and how likely the component matches to that sample $P(G_s = \Phi_w)$.

Update Covariance Matrix for Component Node

Once we update the mean, we will also need to update the covariance matches Σ_a^w for the component node:

$$\Sigma_a^w = \frac{\sum_{s=1}^{S}\sum_{i=1}^{\widehat{G_s}} \overrightarrow{x_i^s}\overrightarrow{x_i^s}^T M_{ia}^{sw} P(G_s = \Phi_w)}{\sum_{s=1}^{S}\sum_{i=1}^{\widehat{G_s}} M_{ia}^{sw} P(G_s = \Phi_w)} \tag{2.13}$$

where $\overrightarrow{x_i^s} = \overrightarrow{N_i^s} - \overrightarrow{N_a^w}$.

Similarly, here we essentially calculated a weighted average of covariance between the component node and sample nodes, $(\overrightarrow{N_i^s} - \overrightarrow{N_a^w})(\overrightarrow{N_i^s} - \overrightarrow{N_a^w})^T$, based on how likely our component node matches to that sample node in match matrix M^{sw} and how likely the component matches to that sample $P(G_s = \Phi_w)$.

Update Mean for Component Edge

Similar to how we calculate the mean for component node $overrightarrowN_a^w$, we calculate the mean for component edge $\overrightarrow{E_{ab}^w}$ as:

$$\overrightarrow{E_{ab}^w} = \frac{\sum_{s=1}^{S}\sum_{i=1}^{\widehat{G_s}}\sum_{j=1}^{\widehat{G_s}} \overrightarrow{E_{ij}^s} M_{ia}^{sw} M_{bj}^{sw} P(G_s = \Phi_w)}{\sum_{s=1}^{S}\sum_{i=1}^{\widehat{G_s}}\sum_{j=1}^{\widehat{G_s}} M_{ia}^{sw} M_{bj}^{sw} P(G_s = \Phi_w)} \tag{2.14}$$

Update Covariance Matrix for Component Edge

Similar to how we calculate the covariance matrix for component node Σ_a^w, we calculate the covariance matrix for component edge Σ_{ab}^w as:

$$\Sigma_{ab}^w = \frac{\sum_{s=1}^{S}\sum_{i=1}^{\widehat{G_s}}\sum_{j=1}^{\widehat{G_s}} \overrightarrow{z_{ij}^s}\overrightarrow{z_{ij}^s}^T M_{ia}^{sw} M_{bj}^{sw} P(G_s = \Phi_w)}{\sum_{s=1}^{S}\sum_{i=1}^{\widehat{G_s}}\sum_{j=1}^{\widehat{G_s}} M_{ia}^{sw} M_{bj}^{sw} P(G_s = \Phi_w)} \tag{2.15}$$

where $\overrightarrow{z_{ij}^s} = \overrightarrow{E_{ij}^s} - \overrightarrow{E_{ab}^w}$.

Delete Redundant Node

Since we initialized the component from sample ARGs, it is likely that the component itself will have background nodes (i.e. nodes not in the common pattern) that are redundant. To reduce the component ARGs from sample ARGs to the smaller common pattern, we can do node deletion based on the node frequency β_a^w and some threshold. For instance, we choose to delete the ath node in Φ_w if:

$$\beta_a^w < 1 - 0.85^n \tag{2.16}$$

where n indicate the nth round of the EM algorithm.

However, you can also choose to used a hard threshold like 0.85 or delete redundant node in the very last round. In addition, we can also choose to delete redundant component here, even though we did not explore such option.

Converge Condition and EM Algorithm Exit

At the beginning of the EM algorithm, the maximization step will make large modification on each parameter, so we would go back to the estimation step after finishing the maximization step.

However, towards the end of the algorithm, we need to check if the EM algorithm has already converged (changes are small). If so, then we can exit the EM algorithm and go to the next step. Here, we simply compared previous component weight $\alpha^{(0)}$ with the current component weight $\alpha^{(1)}$, and exit if the differences is smaller than some number ι:

$$\sum_{w=1}^{W} \alpha_w^{(0)} < \iota \tag{2.17}$$

However, this is not the only choices, and you can use other converging conditions if you like. Especially when your model only contains one component, you will have to use another condition since α_w will always be 1.

2.6 Detect the Spatial Pattern

Once we finished training the model, how do we detect or predict if the extracted/summarized pattern exist in a new ARG? As mentioned above, we can use $f(G|Z)$ calculated by Eq.2.9 as our similarity score to determine if the new ARG, G, contains the same spatial pattern as the training samples. Therefore, we would need to setup some sort of threshold, χ, so that G contains the learned pattern only if $f(G|Z) > \chi$.

To setup χ, we generate a set of R random graphs $\{G^r\}_{r=1}^R$ with similar number of nodes, edge connectivity[2], and labels as our training sample $\{G^s\}_{s=1}^S$. With the random generated ARGs, we can calculate a set of similarity score $\{f(G^r|Z)\}_{r=1}^R$ that can help us determine the threshold χ. If you have enough computational power, and are able calculate the similarity score for a very large number[3]of random ARG. You can simply take the 99.99% percentile of the similarity score set $\{f(G^r|Z)\}_{r=1}^R$.

However, unlike Google[4], we do not live in a magic land. Therefore, we only calculated the similarity score for 50 random ARGs and run z-test on the set of similarity scores $S = \{f(G^r|Z)\}_{r=1}^R$ which allows us to set the threshold χ as $\chi \leftarrow \mu + 3\sigma$ where μ and σ are the mean and variance of the similarity scores of random ARGs.

Due to the application of our model, we do not consider the case where a random but much larger graph could potentially score a higher similarity simply because there are more node to sum. Therefore, if your application are dealing with ARGs on a large spectrum of node number, you might need to modify $f(G|Z)$ to factor in $|G|$.

2.7 Implementation Detail

2.7.1 Coding Complexity

While the model introduced here seems to be pretty straight forward, the actual coding is rather tedious since we have many steps, parameters and nested summation. Therefore, we break down the algorithm and modulated each step so it is easier to debug.

2.7.2 Null Node in the Model

While it is not mentioned above, the model we implemented actually has a null node ϕ, but it does not have any label on the node or edges. Instead, the graph matching algorithm will ignore this null node and used its own null node definition as described in Section 1.7.2. However, when the graph matching return the match matrix and compatibility, M^{sw} and C^{sw}, they include the value for algorithm's own null node which we can used for the model null node. Such implementation allows us to use the graph matching algorithm developed earlier with very little modification and save the troubles of maintaining the parameters for the null node.

2.7.3 Run Time

We trained a model with 2 components on 5 pattern embedded ARG with $\sim$ 30 nodes, and it took $\sim$ 30 minutes including the time for setting up the pattern detection threshold χ. As mentioned in Section 1.5.5, you can adjust the parameter for converging condition and learning rate to tradeoff between accuracy and running time.

[3]How many nodes one node connected to.
[4]Let's say $R > 1,000,000$.
[4]https://twitter.com/deliprao/status/842635509255962624

2.8 Testing the Model

Similar to how we generated test cases in Section 1.6 for graph matching algorithm, we generate a set of ARGs with (G^p) or without (G^r i.e. random ARG) the embedding pattern.

Then we train a model on some of the ARGs with the pattern (i.e. G^p) and plot a histogram of the similarity score ($f(G|Z)$) for all G^p and G^r. This is a sample result generated by a set of ARGs with ~ 30 nodes, and the model is trained by 5 sample ARGs with 2 component ARGs.:

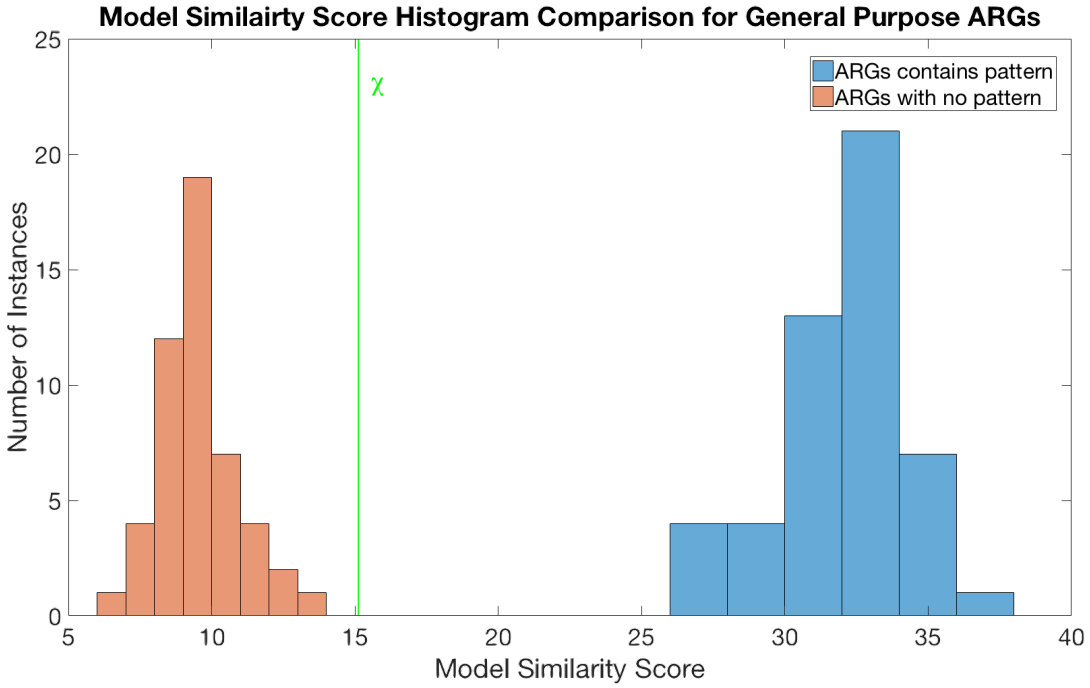

Figure 2.4: *Similarity Score $f(G|Z)$ for pattern embedded ARG (blue) and random ARG(orange) while the green vertical line is the threshold χ.*

2.9 Conclusion

In this chapter we introduced a probabilistic parametric model Z that we can train via EM algorithm. This model allows us to learn the common/share pattern among a set of ARGs, and used a set of component ARGs to represent/summarize such pattern. By calculating $f(G|Z)$ and comparing it to the threshold χ, we can predict if a new ARG G contains the learned pattern.

If we can model a protein crystal structure as an ARG, we can potentially use this model to mine novel protein structure or functional structure from protein crystallography data.

Chapter 3

Protein Modeling

3.1 Introduction

Proteins are macromolecules responsible for nearly every task of cellular life. They are 3D structures consisting of amino acid sequences translated from genes and interact with each other to carry out essential functions, such as catalyzing metabolic reaction, transport molecules, respond to stimuli, and so on. Such interactions often happen at sites (e.g. protein domains) that are conserved between proteins across species. By recombining and rearranging these domains as proteins' basic building block, molecular evolution is able to create proteins with different functions.

Conventional computational methods for studying protein domains between proteins mainly use the sequences and secondary structures of proteins to do pattern matching, which ignores the 3D nature of proteins and cannot take advantage of the available 3D structures of proteins. For instance, the beginning and the end of an amino acid sequence might come together as a single function group after folding. However, sequential pattern matching would not be able to capture such union. and would produce two separate patterns (one at the beginning and one at the end), instead of a single functional pattern. Therefore, to achieve better understandings about protein interactions and their functions, we need to investigate the detailed spatial characteristics of the 3D structures of proteins.

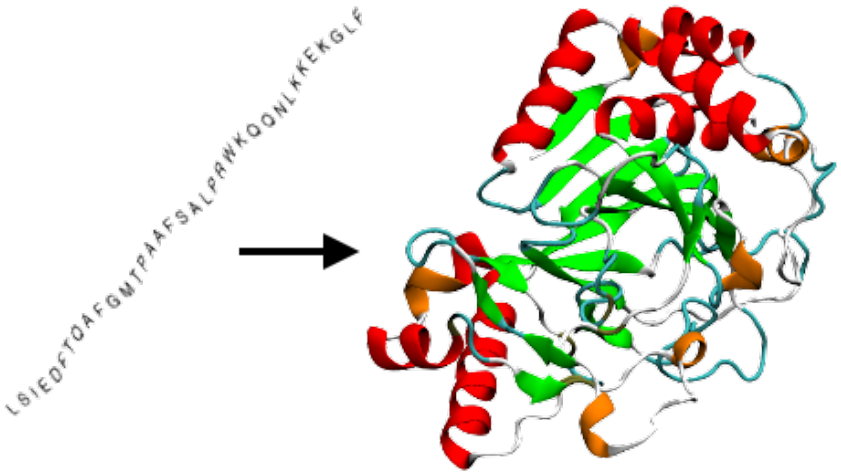

Figure 3.1: *Conventional computational methods relies sequential data while protein functions in its 3D structure where amino acids that do not appear sequentially could come together to form pattern that are hard to discover in its sequential form.*

Under such realization, many researchers have started manually matching the 3D structures of proteins, but this is very time consuming and can be subjective. Therefore, if we could generate a good ARG representation for protein 3D structure from crystallography data, our pattern learning could be a great tool for extracting share structure among proteins.

3.2 Protein to ARG

To apply the model on protein crystallography data, we will first need to build a good ARG representation for the protein 3D structure. Protein crystallography data usually comes in as a PDB(Protein Data Bank) format and provides the 3D coordinate (x, y, z) of each amino acid. We turn the 3D structure into a protein ARG where nodes represent different amino acid and edges represent the distance between these amino acids.

3.2.1 Edge for Protein ARG

For the edge protein ARG, we assign it a single scalar which is the euclidian distance of two amino acid:

$$\overrightarrow{E_{ab}} = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2 + (z_a - z_b)^2} \tag{3.1}$$

However, we don't want to create an edge for every pair of amino acids. Therefore we choose an $8\mathring{A}$ as our distance cutoff, and only create edge for amino acid pairs whose euclidian distance is less than $8\mathring{A}$. Here is the histogram for pairwise distance between two amino acids:

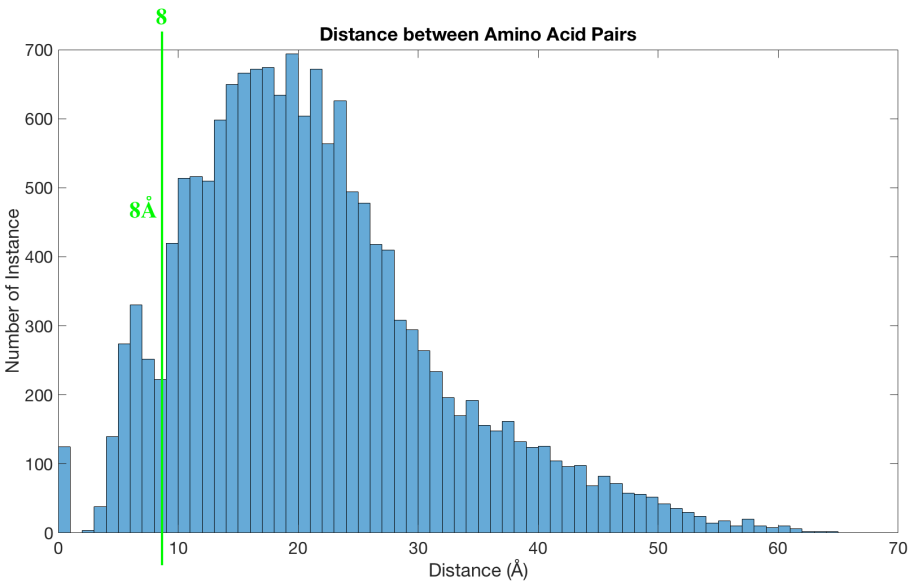

Figure 3.2: *As expected, we can roughly see a mixture of two normal distribution where the one with the smaller mean representing the distance between two related amino acids while the one with larger mean representing the distance between two random/unrelated amino acids.*

Interestingly, even though we pick the $8\mathring{A}$ cutoff purely based on the distance distribution, a single amino acid also has a diameter $\sim 8\mathring{A}$.

Since $\overrightarrow{E}$ is just a scalar following a normal distribution, we simply used the assume Gaussian PDF to calculate the compatibility and update the mean as well as the variance.

3.2.2 Node for Protein ARG

While setting up the protein ARG edge is pretty straight forward, creating a reasonable representation and compatibility function is very difficult, and here are a couple of options:

One Hot Representation

The simplest representation for amino acid would just be indexing the 20 amino acids from 1 to 20 and set $\overrightarrow{N}$ to the associated amino acid index. Our compatibility therefore would become:

$$C_{ai} = \begin{cases} 1 & \overrightarrow{N_a} = \overrightarrow{N_i} \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

However, such compatibility function is not biologically accurate because some amino acids might share similar biochemistry property so that they can sometime substitute each other without changing the overall structure of the function unit. Therefore, even though two amino acids have two different name, they can still have some similarity and be compatible in some context. The simple one hot representation in this case does not work.

BLOSUM Matrix

Since amino acids that share similar biochemistry property might be compatible in some cases, we also consider using BLOSUM matrix in our compatibility function while keeping the one hot index representation for each node $\overrightarrow{N}$.

BLOSUM matrices were first introduced by Steven Henikoff and Jorja Henikoff in 1989. They scanned a protein database with align protein sequence and counted the relative frequencies of amino acids and their substitution probabilities. Then, they calculated a log-odds score and produced a substitution matrix B for the 20 standard amino acids:

	A	B	C	D	E	F	G	H	I	K	L	M	N	P	Q	R	S	T	V	W	X	Y	Z
A	4	-2	0	-2	-1	-2	0	-2	-1	-1	-1	-1	-2	-1	-1	-1	1	0	0	-3	-1	-2	-1
B	-2	6	-3	6	2	-3	-1	-1	-3	-1	-4	-3	1	-1	0	-2	0	-1	-3	-4	-1	-3	2
C	0	-3	9	-3	-4	-2	-3	-3	-1	-3	-1	-1	-3	-3	-3	-3	-1	-1	-1	-2	-1	-2	-4
D	-2	6	-3	6	2	-3	-1	-1	-3	-1	-4	-3	1	-1	0	-2	0	-1	-3	-4	-1	-3	2
E	-1	2	-4	2	5	-3	-2	0	-3	1	-3	-2	0	-1	2	0	0	-1	-2	-3	-1	-2	5
F	-2	-3	-2	-3	-3	6	-3	-1	0	-3	0	0	-3	-4	-3	-3	-2	-2	-1	1	-1	3	-3
G	0	-1	-3	-1	-2	-3	6	-2	-4	-2	-4	-3	0	-2	-2	-2	0	-2	-3	-2	-1	-3	-2
H	-2	-1	-3	-1	0	-1	-2	8	-3	-1	-3	-2	1	-2	0	0	-1	-2	-3	-2	-1	2	0
I	-1	-3	-1	-3	-3	0	-4	-3	4	-3	2	1	-3	-3	-3	-3	-2	-1	3	-3	-1	-1	-3
K	-1	-1	-3	-1	1	-3	-2	-1	-3	5	-2	-1	0	-1	1	2	0	-1	-2	-3	-1	-2	1
L	-1	-4	-1	-4	-3	0	-4	-3	2	-2	4	2	-3	-3	-2	-2	-2	-1	1	-2	-1	-1	-3
M	-1	-3	-1	-3	-2	0	-3	-2	1	-1	2	5	-2	-2	0	-1	-1	-1	1	-1	-1	-1	-2
N	-2	1	-3	1	0	-3	0	1	-3	0	-3	-2	6	-2	0	0	1	0	-3	-4	-1	-2	0
P	-1	-1	-3	-1	-1	-4	-2	-2	-3	-1	-3	-2	-2	7	-1	-2	-1	-1	-2	-4	-1	-3	-1
Q	-1	0	-3	0	2	-3	-2	0	-3	1	-2	0	0	-1	5	1	0	-1	-2	-2	-1	-1	2
R	-1	-2	-3	-2	0	-3	-2	0	-3	2	-2	-1	0	-2	1	5	-1	-1	-3	-3	-1	-2	0
S	1	0	-1	0	0	-2	0	-1	-2	0	-2	-1	1	-1	0	-1	4	1	-2	-3	-1	-2	0
T	0	-1	-1	-1	-1	-2	-2	-2	-1	-1	-1	-1	0	-1	-1	-1	1	5	0	-2	-1	-2	-1
V	0	-3	-1	-3	-2	-1	-3	-3	3	-2	1	1	-3	-2	-2	-3	-2	0	4	-3	-1	-1	-2
W	-3	-4	-2	-4	-3	1	-2	-2	-3	-3	-2	-1	-4	-4	-2	-3	-3	-2	-3	11	-1	2	-3
X	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
Y	-2	-3	-2	-3	-2	3	-3	2	-1	-2	-1	-1	-2	-3	-1	-2	-2	-2	-1	2	-1	7	-2
Z	-1	2	-4	2	5	-3	-2	0	-3	1	-3	-2	0	-1	2	0	0	-1	-2	-3	-1	-2	5

Figure 3.3: *Letter here represents different amino acids.*

And the compatibility function would then become:

$$\overrightarrow{C_{ai}} = B[\overrightarrow{N_a}, \overrightarrow{N_i}] \tag{3.3}$$

While the BLOSUM matrix B address the problematic compatibility function for one hot representation. Another problem that we would need to address is model node update. If you recall how we update the mean for component node in E.q. 2.12, we took a weighted average of all the node based on how the node matching probability and sample matching probability. However, since the index is not continuous, it doesn't really make sense to take the mean any more. During our trail, we simply take the index from the node that has the highest weight/matching probability. This is certainly not ideal, and we want to search for a continuous representation for amino acid that are continuous with a compatibility function that consider the similar biochemistry property of different amino acids.

Amino Acid 2 Vec

In natural language processing, researchers deal with a similar problem for one hot representation of word. They came up with word embedding based on its surrounding context, and developed a very effective model, Skip-Gram, for generating word vector. With library like Gensim[1], we can generate our own word vector by simply feeding the model enough document with words.

If we think of amino acid sequences as documents and individual amino acid as word, can we generate amino acid vector the same way using model like Gensim? It makes sense because amino acids share similar biochemistry property are likely to stay in a similar chemistry environment created by nearby amino acids. We feed the Skip-Gram model with protein sequences across many species, and get protein vectors $\overrightarrow{AA}$ with a dimension of 20[2]that cluster by their biochemistry property:

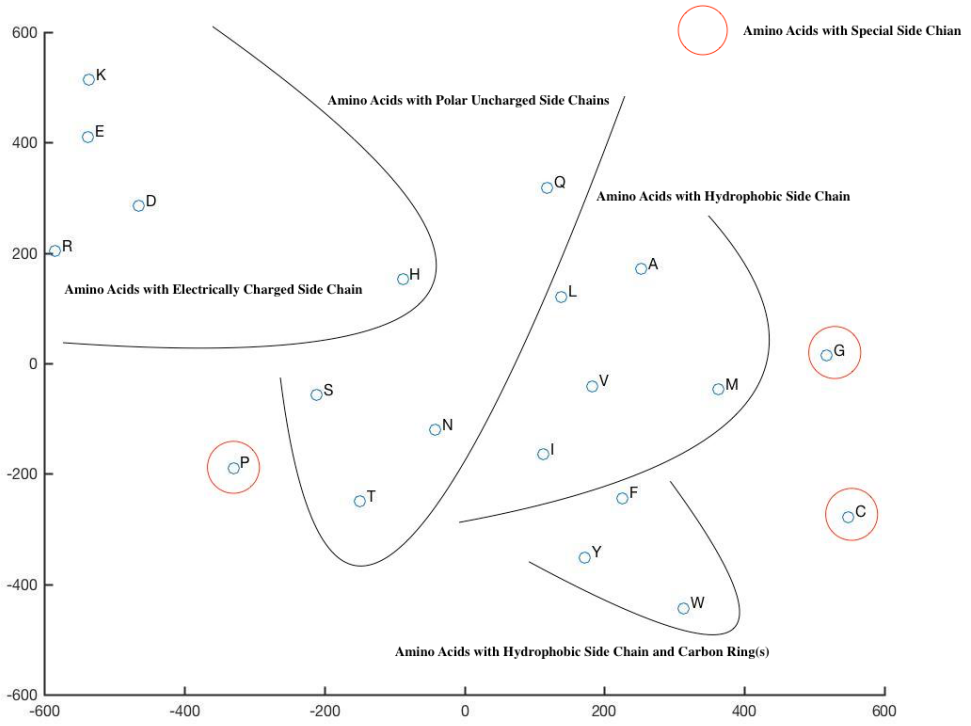

Figure 3.4: *Amino Acid Vector* $\overrightarrow{AA}$ *clusters after dimension reduction (PCA) from* 20 *to* 2.

[1]https://radimrehurek.com/gensim/
[2]Since we have 20 amino acids.

Therefore, we can label each node with its associated amino acid vector and our compatibility function would just be the euclidian distance of the two amino acid vector:

$$\overrightarrow{C_{ai}} = \sqrt{\sum_{z=1}^{20} (\overrightarrow{N_a}[z] - \overrightarrow{N_i}[z])^2} \tag{3.4}$$

We also construct a 20×20 compatibility matrix for the 20 amino acids, and it follow a similar form as the BLOSUM matrix after scaling. For model node, updating mean is just as easy as follow E.q. 2.12.

This is a very cool idea with lots of potential, but the drawback of such representation would be the lack of locality. Even though our amino acid vector can be clustered by their biochemistry property, they are likely to show slightly different property in different environment, so using a global vector to represent amino acid in different context might not be ideal as well.

Local Substitution Vector

BLOSUM matrix is a brilliant idea as we can calculate the amino acid substitution odd based on protein alignment database. Can we use a similar idea to factor in the locality of the amino acid and solve the model node update problem described above all together?

Within the sample protein ARG, each node will have a single amino acid index as their $\overrightarrow{N^s}$. However, for the component protein ARG, the model node a would have a local substitution vector $\overrightarrow{N^w}$ representing the odd for this node to be matched with a specific amino acid. Therefore, the compatibility function would become:

$$\overrightarrow{C_{ai}^{sw}} = \overrightarrow{N_i^w}[\overrightarrow{N_a^s}] \tag{3.5}$$

Since $\overrightarrow{N^w}$ is continuous substitution vector, we now can easily update the mean for model node update following E.q. 2.12. But what value should we used for $\overrightarrow{N^w}$, and how should we update it?

When initializing the model protein ARG, we set the local substitution vector based on the BLOSUM matrix, so $\overrightarrow{N_i^w} = B[AA_i, :]$, where AA_i is the amino acid index for node i. To update the substitution vector with locality, we first calculate a "weighted" frequency vector $\overrightarrow{F_i^w}$:

$$\overrightarrow{\zeta_i^w}[z] = \sum_{s=1}^{S} \sum_{a}^{\{a|\overrightarrow{N_a^s}=z\}} M_{ai}^{sw} P(G_s = \Phi_w) \qquad \forall z = 1, 2, 3...20 \tag{3.6}$$

$$\overrightarrow{F_i^w}[z] = \frac{\overrightarrow{\zeta_i^w}[z]}{\sum_{x=1}^{20} \overrightarrow{\zeta_i^w}[x]} \qquad \forall z = 1, 2, 3...20 \tag{3.7}$$

Once we calculate $\overrightarrow{F_i^w}$, we can calculate the log odd-ratio with an amino acid background frequency $\overrightarrow{F_B}$ that we can pre-defined or calculate based on the sample protein ARGs:

$$\overrightarrow{N_i^w}[z] = \log(\frac{\overrightarrow{F_i^w}[z]}{\overrightarrow{F_B}[z]}) \qquad \forall z = 1, 2, 3...20 \tag{3.8}$$

While this seems complicated, but it is the same idea as BLOSUM matrix. Instead of using a conserve protein alignment database as for BLOSUM, we used our own alignment/matching matrix M^{sw}. Since M^{sw} is computed both by amino acid compatibility as well as structure/edge compatibility, we are able to introduced locality into such local substitution vector.

3.3 Protein Backbone Encoding in Objective Function

During the development, we observe matching result where a couple nearby amino acids in one ARG being matched to amino acids that are far apart in another ARG due to similar environment (the first half of E.q. 1.1).

However, it's not very likely[3]that three immediate neighbor amino acids in one protein match to entirely different places in another protein. That's because, even though protein can fold, it still has a peptide chain that serve as its backbone that you can't stretch or turn too much. Therefore, we modified the energy/objective function during graph learning E.q. 1.1, and introduced a new term to encode the constraints brought by the protein backbone:

$$E(M) = -\frac{1}{2}\sum_{a=1}^{G}\sum_{i=1}^{G'}\sum_{b=1}^{G}\sum_{j=1}^{G'} M_{ai}M_{bj}C_{abij} - \alpha\sum_{a=1}^{G}\sum_{i=1}^{G'} M_{ai}C_{ai}$$
$$-\beta\sum_{a=1}^{G}\sum_{i=1}^{G'} M_{a-1i-1}M_{ai}M_{a+1i+1} \tag{3.9}$$

While the addition looks complicated, it is simply saying that: if node a in G has a good match with node i in G', than the node immediately before and after node a should also have a good match with the node immediately before and after node i as we are indexing the node by their sequence order.

3.4 Proof of Concept

Just to do a proof of concept[4], we trained a model on 5 protein structure all contains same part of the *CH1* domain with 2 components and test it on the original structure, mutated structure(reversing the entire sequence, partition the sequence and reverse the order of partition) that are impossible to detect without 3D structure information, and finally random structure:

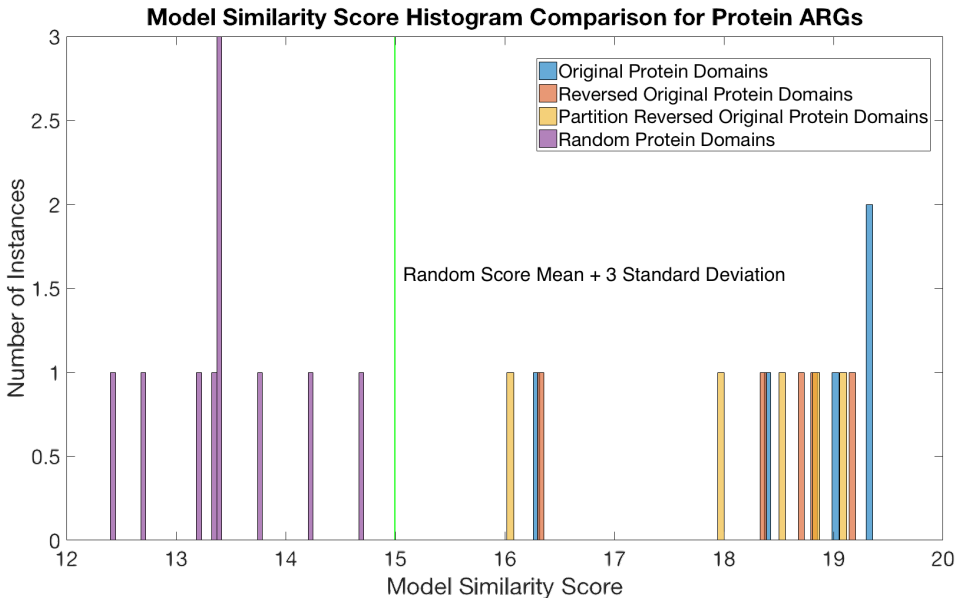

Figure 3.5: *Model similarity score ($P(G|Z)$) for different test cases.*

[3]Though not impossible.

[4]Not enough time to do it all!

As you can see, our trained model can clearly separate *CH1* domain from random structure. Moreover, for cases where the sequence(indexes) is entirely reversed or reversed in partition (i.e. part of structure move to another place), model are still able to capture the domain. However, if we used traditional sequential matching techniques to find the *CH1* domain, we won't be able to recover the domain if the sequence is reversed or reversed by partition. In addition, our graph matching algorithm also operates in very high precision thanks to the local substitution vector. Here are some examples of protein matching result visualizing in their 3D form:

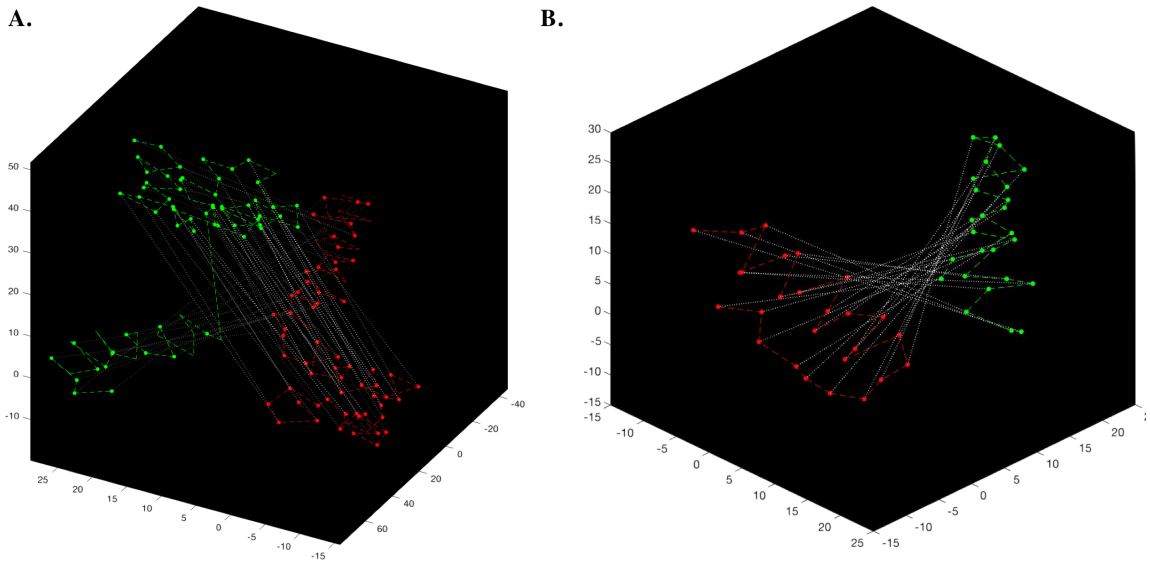

Figure 3.6: *Two examples of amino acids from two different protein (green and red) are matched (white) based on the graph matching matrix M.*

3.5 Conclusion

In this chapter we introduced many ways that we can model the protein 3D structure as a ARG. Then, we show a proof of concept about training a model to learn and summarize part of the *CH1* domain. It shows us how such model can take advantage of the 3D structure data, and recover for sequence changes. Invariance to sequential changes, our model shows potential for discover folded structure and novel motif that are overlooked by traditional sequential alignments.

Chapter 4

Looking Forward

4.1 Graph Matching

4.1.1 More Efficient Implementation

While the graph matching algorithm is already an approximation, it is still not fast enough to support the pattern learning model to learn in very large scale. Therefore, a more efficient implementation is of interest from an engineering view point and if we want to move the algorithm to production system.

One major time sink of the algorithm is the iterative row and column normalizations enforcing by the two-way constraints. A potential solution would be using GPU to perform row/column normalization independently in parallel fashion. Another solution would be ignoring the normalizations altogether, and incorporating the two-way constraints into our objective functions with Lagrange multipliers:

$$\begin{aligned} E(M) = &-\frac{1}{2}\sum_{a=1}^{G}\sum_{i=1}^{G'}\sum_{b=1}^{G}\sum_{j=1}^{G'} M_{ai}M_{bj}C_{abij} \\ &-\frac{1}{\beta}\sum_{a=1}^{G}\sum_{i=1}^{G'} M_{ai}(\log M_{ai} - 1) \\ &+\sum_{a=1}^{G}\mu_a(\sum_{i=1}^{G'} M_{ai} - 1) + \sum_{i=1}^{G'}\nu_a(\sum_{a=1}^{G} M_{ai} - 1) \end{aligned} \tag{4.1}$$

and we can directly derive M via the objective function using library like Theano[1]. However, because of the nested sum, even though there is a matrix operation that can help us do the forward calculation, the back-propagation/derivation has not been implemented yet. Therefore, we did not implement such solution, but it would be much more efficient once the derivation for such operations are implemented.

Another time and memory sink for the algorithm would be the compatibility between edges. While the computation can be speed up via parallel computing, there are lots of communication overhead due to the design of the edge compatibility matrix, where each row or column is associated with a single edge in the graph and result in a $(|G| * |G|) \times (|G'| * |G'|)$ matrix. Since MATLAB only allows parallel computing row by row or column by column, we are still copying a huge vector in parallel task which result in lots of communication overhead. In addition, the

[1] http://www.deeplearning.net/software/theano/

compatibility matrix is very huge and could result in memory issue. Even though we can fix by storing it as a sparse matrix, sparse matrix could also introduce lots of computation overhead during the graduated assignment process.

Therefore, it would be beneficial for us to think of another representation for edge compatibility scores that are both memory and computation efficient.

4.1.2 Automate Parameters Tuning

A lot of the frustrations during the development of this algorithm coming from tuning the parameter considering the large number of different combinations of parameter. Considering different application of the same algorithm could have very different optimal parameters configuration, it would be beneficial for users if the algorithm has some mechanisms to tune the parameter automatically based on the expected matching result.

While this problem can easily turn into another project about model learning, we can start from a simple grid search, or learn about how different parameters could affect different aspects of the matching result and adjust them accordingly.

4.2 Pattern Learning

4.2.1 Smarter Component Initiation

In our model, the initial components can have a huge impact on model's pattern learning quality. For instance, if the share pattern has two components but our starting components only have one of them, the model could never capture the entire pattern. Therefore, we might want to be smarter than random when initializing our model components.

One potential solution would be do a pairwise graph matching before picking the components, and incorporate some heuristic rules (e.g. matching result clarity, larger matching nodes etc) to help us pick the components.

Another solution would be introducing a mechanism allows the model to swap in random sample ARGs as new components, and switch it our if the performance does not get better.

4.2.2 Component and Node Deletion

Since we start off with sample ARGs as our components ARGs, it is important for the components ARGs to reduce themselves in order to accurately represent the share pattern. However, in many of our experiments, the node deletion is not always perfect just based on a threshold. Therefore, we might want to introduce more heuristic rules for deletion in the future. One such rule could be examining β^w across multiple rounds, and only delete it based on a history of pattern.

In addition to node deletion, it would be helpful if we could delete the entire component as well since the randomly initialized components could share similar pattern, and therefore become redundant. Introducing a mechanism for component deletion not only allow us to start from a relatively large pool of components, but also help us speed up the model training. One potential way we could do this is by comparing the result of two components matching to the same sample ARG.

4.2.3 Other Applications

Besides protein structure mining, another interesting application for the pattern learning model would be computer vision, which is dominated by Neural Network nowadays.

Neural Network like CNN(Convolutional Neural Network) is a very effective model for object recognition because the non-linear transformation and back-propagation are able to learn features automatically and effectively. Moreover, pooling layer has been effective in handling minor variance while different filters can handle large variance like rotation[2]. However, when the scene becomes more complicated, describing the relationship between multiple object for instance, CNN becomes less effective (larger model and more training data) because CNN is more effective in learning features, but less effective in modeling relationship. For instance, if we have two objects whose relationship is a certain distance (A), to recognize the same relationship after rotation (B), a CNN might need to learn an additional filter for the 45° angle, while an ARG representation can easily recognize the rotated scene with no trouble:

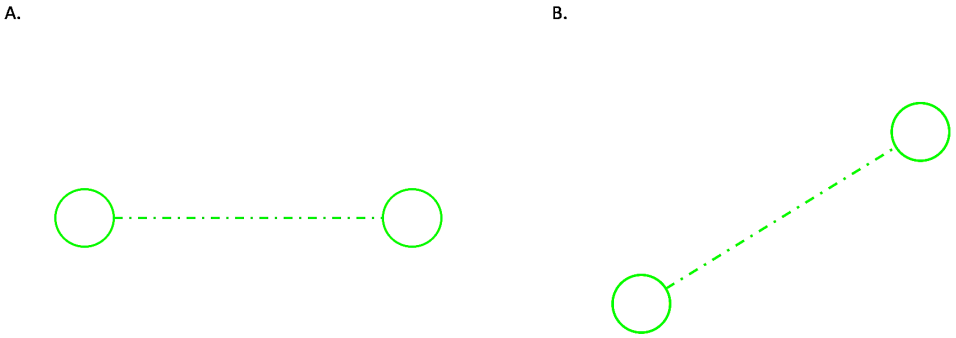

Figure 4.1: *A. Two objects (circles) have a relationship of a certain distance X. B. A rotated scene showing such relationship.*

Therefore, the pattern learning model described here can be very helpful in understanding the relationship between complex scene ("student taking class", "cars stuck in traffic" etc.) while CNN helps the model to recognize individual object in the scene or generate latent representation for individual object.

Besides computer vision, the pattern learning model can also be used in natural language processing task, like modeling the relationship (e.g. Q&A, extension, agreement, etc.) between sentences in a dialogues. Here, we can turn each word/concept to a node (whose label can be the word vector) and the word/concept distance as the edge. Once we turn the conversation exchange into an ARG, we might be able to model different relationship as well.

Therefore, combining neural network for individual object and pattern learning for modeling relationship, machines probably can learn complex structure, scenes, conversations and many other things more effectively. I am really excited about what's coming next.

[2]Filters might not be representing different rotations exactly but rotation certainly requires more filters to summarize more features.

4.3 Protein Modeling

4.3.1 Novel Structure Motif in Protein

While the proof of concept we did in Section 3.4 is nice, domain discovery is not a very challenging task. Due to its relatively large size, simple 3D alignment should give you a reasonably good match and you don't really need to model the protein as an ARG.

However, if we are able to train the model efficiently with enough data, we might be able to discover 3D structure motifs that have long been overlooked by sequential matching algorithm. If the model can learn such novel motif, it would be of great help for many biologists.

4.3.2 Protein as Documents and Amino Acid as Word

In Section 3.2.2, we treated protein as documents and amino acid as word, which allows us to use model in natural language field to generate representation for amino acid.

In the same line, many of the models developed today in the natural language field, especially the kinds dealing with sequential model, might be a great tool for answering question about protein. For instance, we can use a Seq2Seq model to predict structure of protein sequence, or use LSTM model to predict protein chemical property with the protein sequence.

4.3.3 Edge for Protein ARG Revisit

While tuning hyper parameters for the model, we realized that the node compatibility needs to play a more important role(larger α) in order to get clear and correct result for graph matching. This makes sense because protein alignment should be driven more by the amino acid compatibility. However, can we give edge here a different representation and a more effective relationship to help with matching/alignment?

Appendix

Here are the code and presentation for the thesis.

Graph Matching Implementation Code:
https://github.com/WesleyyC/Graph-Matching

Pattern Learning Implementation Code:
https://github.com/WesleyyC/Spatial-Pattern-Learning

Protein Modeling Implementation Code:
https://github.com/WesleyyC/Proteins-Domain-Learning

Thesis Presentation:
https://docs.google.com/presentation/d/1ql-HDbQCY-ff4cXDc2qcd4sy0qTivMSxWtxJGo5Kcxo/edit?usp

Bibliography

[1] Steven Gold and Anand Rangarajan. "A Graduated Assignment Algorithm for Graph Matching". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 18.4 (Apr. 1996), pp. 377–388. ISSN: 0162-8828. DOI: 10.1109/34.491619. URL: http://dx.doi.org/10.1109/34.491619.

[2] Pengyu Hong and Thomas S. Huang. "Spatial Pattern Discovery by Learning a Probabilistic Parametric Model from Multiple Attributed Relational Graphs". In: *Discrete Appl. Math.* 139.1-3 (Apr. 2004), pp. 113–135. ISSN: 0166-218X. DOI: 10.1016/j.dam.2002.11.007. URL: http://dx.doi.org/10.1016/j.dam.2002.11.007.

[3] Richard Sinkhorn. "A Relationship Between Arbitrary Positive Matrices and Doubly Stochastic Matrices". In: *The Annals of Mathematical Statistics* 35.2 (1964), pp. 876–879. ISSN: 00034851. URL: http://www.jstor.org/stable/2238545.