Brandeis University

Senior Thesis in Computer Science

# Graph Matching, Pattern Learning, and Protein Modeling

*Wei Qian*

Supervised by
Prof. Pengyu Hong

May 2, 2017

# Contents

# 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.

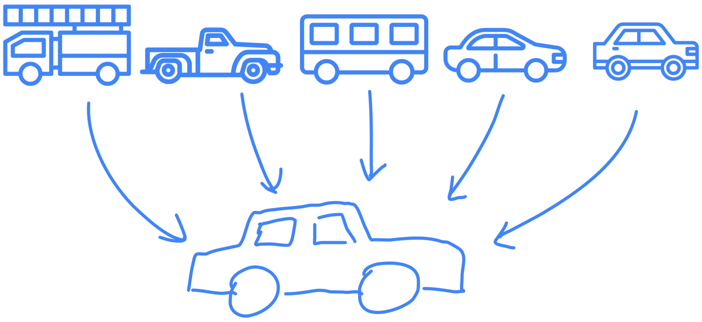


Figure 1: *Different cars can be summarized as some simple components in a certain spatial relationship.*[1]

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 self-linked null node netowrk to each ARG to avoid matches among nodes that are 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, 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.

Prof. Hong gave me a shot in the summer of 2015, and asked me to implement some preexisting algorithms in MATLAB. We then modified the algorithm to handle some issues occurred while processing larger scale data, and 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 drawed me to computer science 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 the 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 even more and inspired me to do great work, 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 $a$th node in the graph) and edges ($\overrightarrow{E_{ab}}$ is the label for the edge from the $a$th node to the $b$th 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, $\phi$, 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 $a$th node in $G$ to the $i$th 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 $\forall i \neq \phi$, $\sum_{i=1}^{G} M_{ai} = 1$:

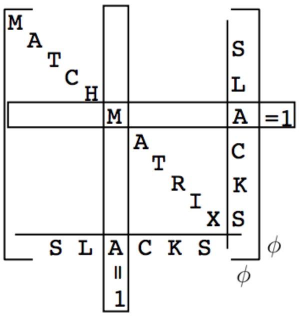


Figure 1.1: *Both rows and columns should sum up to 1 except the row and column involving $\phi$ nodes, which are considered the slack row and slack column.*

## 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

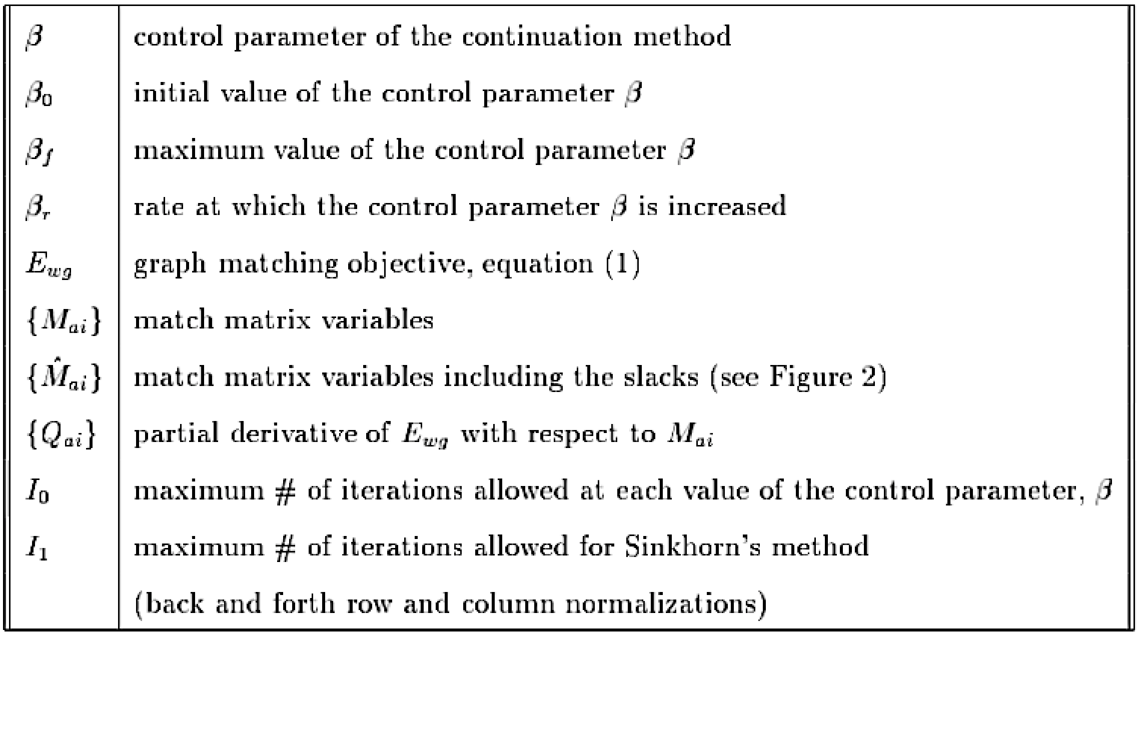
$$+\sum_{a=1}^{G}\sum_{i=1}^{G'} Q_{ai} M_{ai} \tag{1.6}$$

which is an assignment problem!

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

```
Initialize $\beta$ to $\beta_0$, $M_{ai}$ 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:



| | |
|---|---|
| $\beta$ | control parameter of the continuation method |
| $\beta_0$ | initial value of the control parameter $\beta$ |
| $\beta_f$ | maximum value of the control parameter $\beta$ |
| $\beta_r$ | rate at which the control parameter $\beta$ 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, $\beta$ |
| $I_1$ | 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 $\delta$ is the covariance matrix and $\zeta$ is the dimension of $\overrightarrow{a}$ and $\overrightarrow{b}$. Since we assume feature independence, the covariance matrix $\delta$ 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 $\iota$:

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

### 1.5.5 Run Time

The algorithm takes $\sim 40s$ to complete when matching two graphs with $\sim 40$ nodes. However, you can certainly adjust the parameter like $I$ and $\beta$ 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 $\phi$, 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 $\beta$ and $\beta_r$ 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:

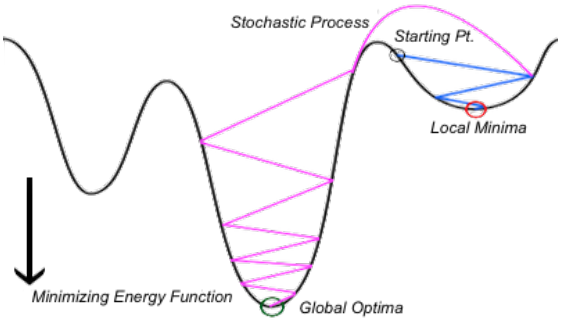


Figure 1.2: *The intuition behind introducing the stochastic process.*

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

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

where $\tau$ 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** $\beta$ to $\beta_0$, $M_{ai}$ 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 $> I_0$)
    $M^0_{ai} \leftarrow M^0_{ai} + \frac{\tau * U(-1,1)}{|G|}$
    $Q_{ai} \leftarrow \sum_{b=1}^{G} \sum_{j=1}^{G'} M^0_{bj} C_{abij} + \alpha C_{ai}$
    $M^0_{ai} \leftarrow exp(\beta Q_{qi})$
        **Begin** $C$**:** (Do $C$ until $M$ converges or # of iterations $> I_1$)
        Update $M$ by normalizing across all rows:
        $M^1_{ai} = \frac{M^0_{ai}}{\sum_{i=1}^{G'} M^0_{ai}}$ for all $a \neq \phi$
        Update $M$ by normalizing across all columns:
        $M^0_{ai} = \frac{M^1_{ai}}{\sum_{a=1}^{G} M^1_{ai}}$ 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:

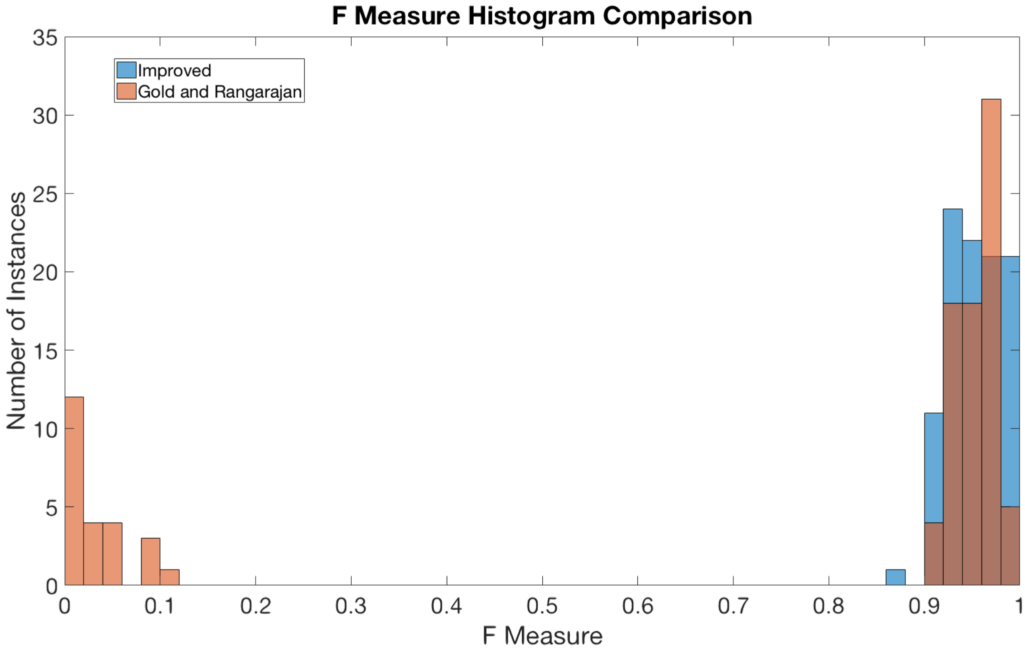


Figure 1.3: *$F_1$ score distribution histogram for the original algorithm (orange) and the improved algorithm with stochastic process (blue).*

### 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 $\phi$ in $G'$. We realized that since the original algorithm define the compatibility function so that any compatibility related to the $\phi$ node will be set to 0. Therefore, the algorithm does not *actively* match background nodes to $\phi$ node, but *hope* that the background noise does not attract to one specific node, and end up matching with the $\phi$ 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:

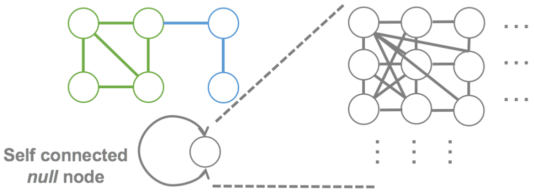


Figure 1.4: *The self linked/connected null node $\phi$ (grey) can be treated as a fully connected null node that can be matched to any pattern including background nodes.*

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):

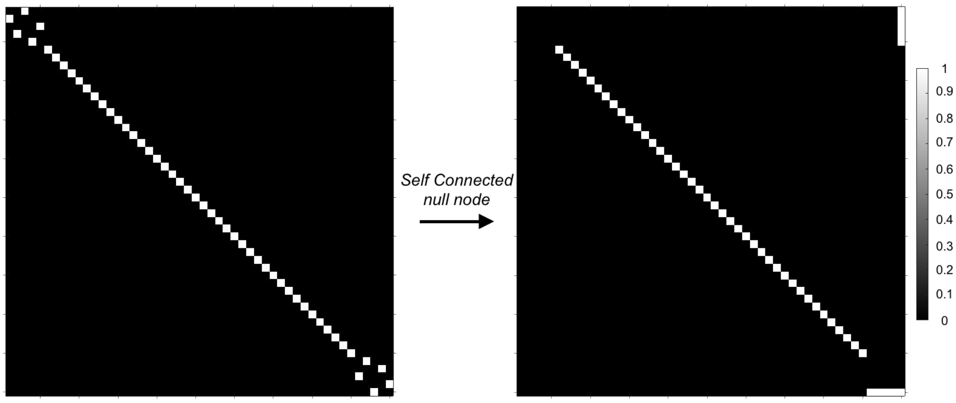


Figure 1.5: *This is a visualization of the match matrixes produced by the original algorithm by Gold and Rangarajan (left) and the modified version (right).*

## 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:

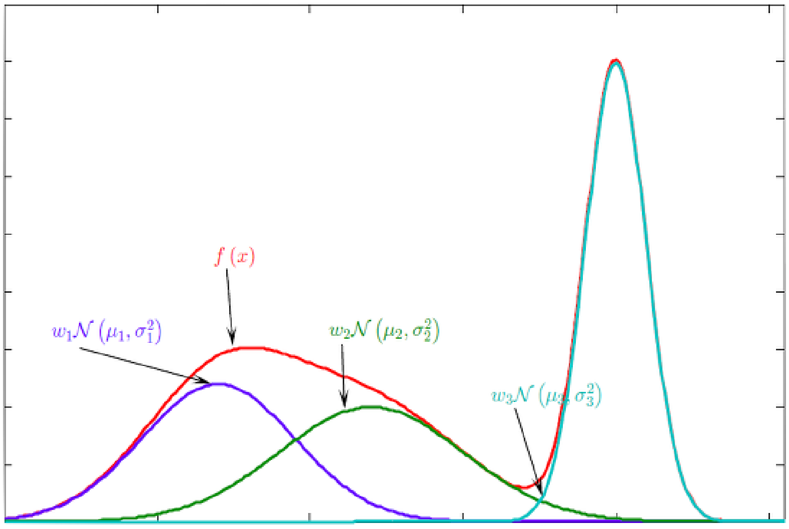


Figure 2.1: *The data distribution generated by $f(x)$ can be captured by three normal distribution with various means and variances.*

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:

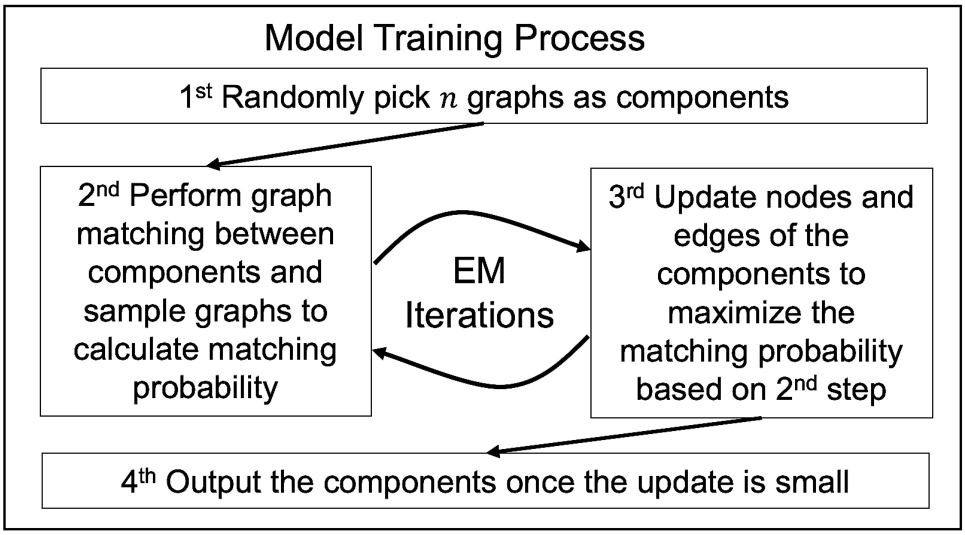

Model Training Process
1st Randomly pick $n$ graphs as components
2nd Perform graph matching between components and sample graphs to calculate matching probability
EM Iterations
3rd Update nodes and edges of the components to maximize the matching probability based on 2nd step
4th Output the components once the update is small

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 $\alpha_w$ 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 $\Sigma_a^w$ and $\Sigma_{ab}^w$ with a similar index. Last but not least, for each node in each component, there is an associated frequency/weight $\beta_a^w$ 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 $\Phi_w$ 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 $\alpha$, mean for each node/edge $\overrightarrow{N^w}/\overrightarrow{E^w}$, and the covariance associated with them $\Sigma$.

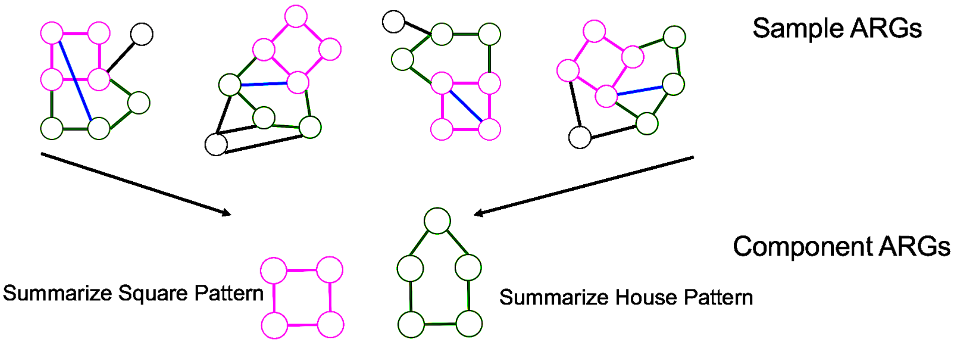


Figure 2.3: *Component can summarize different part of the common pattern.*

## 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 $\Phi$. 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 $\Phi_w$, we would have:

$$\begin{align}
\overrightarrow{N_a^w} &\leftarrow \overrightarrow{N_a^s} & \forall a \in \widehat{G_s} \tag{2.1}\\
\overrightarrow{E_{ab}^w} &\leftarrow \overrightarrow{E_{ab}^s} & \forall a, b \in \widehat{G_s} \tag{2.2}\\
\Sigma_a^w &\leftarrow I \tag{2.3}\\
\Sigma_b^w &\leftarrow I \tag{2.4}\\
\alpha_w &\leftarrow \frac{1}{W} \tag{2.5}\\
\beta_a^w &\leftarrow \frac{1}{|G_s|} & \forall a \in G_s) \tag{2.6}
\end{align}$$

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 $\Phi_w$ 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 $\Phi_w$:

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

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

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 $\alpha_w$ for each component $\Phi_w$ 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 $\Phi_w$:

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

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 $\Sigma_a^w$ 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 $\Sigma_a^w$, we calculate the covariance matrix for component edge $\Sigma_{ab}^w$ 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 $\beta_a^w$ and some threshold. For instance, we choose to delete the $a$th node in $\Phi_w$ if:

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

where $n$ indicate the $n$th 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 $\iota$:

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

However, these are not the only choices, and you can use other converging conditions if you like.

## 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, $\chi$, so that $G$ contains the learned pattern only if $f(G|Z) > \chi$.

To setup $\chi$, 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 $\chi$. 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 $\chi$ as:

$$\chi \leftarrow \mu + 3\sigma$$

where $\mu$ and $\sigma$ are the mean and variance of the similarity scores of random ARGs.

## 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 $\phi$, 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 $\sim 40$ nodes, and it took $\sim TBD$ minutes.

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 $G^r$:

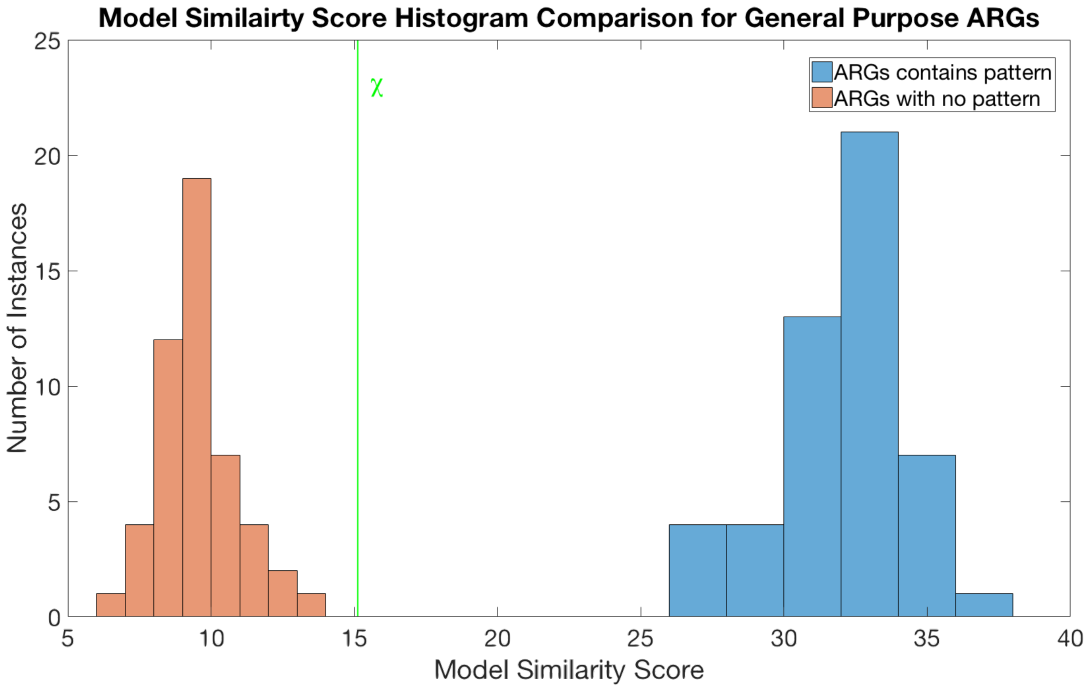


Figure 2.4: *Similarity Score $f(G|Z)$ for pattern embedded ARG (blue) and random ARG(orange) while the green vertical line is the threshold $\chi$.*

## 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 $\chi$, 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

# Chapter 4

# Further Questions

## 4.1 Reconstruction Hypothesis

The Reconstruction Hypothesis is a claim in theoretical computer science that is very likely true, but has not been proven to be. The hypothesis revolves around the notion of a 'deck' of a graph $G$ with $N$ vertices. The Deck of G ($Deck(G)$) is a (possibly multi-)set of $N$ graphs, each of size $N-1$, all of which are created by deleting one vertex from the graph G.

Two Decks $D_1$ and $D_2$ are said to be isomorphic if there exists a one to one mapping of their elements such that every graph $g_i \in D_1$ is mapped to an isomorphic graph $g_j \in D_2$. The reconstruction hypothesis is the claim that two decks are isomorphic if and only if their graphs are isomorphic, or in formal terms:

$$Deck(G) \cong Deck(H) \iff G \cong H$$

Though this hypothesis seems pretty iron clad, there is one known violation, two non-isomorphic graphs over two vertices have congruent decks. Can you find them? (hint: there are only two graphs over two vertices). Thus the actual hypothesis only makes its claim for $N \geq 3$. As we get to larger and larger values of $N$, it would seem that the hypothesis becomes harder to refute via a counter example, both because the amount of information that goes into the physical representation of the deck increases cubically, and because the unknown pieces of the graph shrink relative to the consistent amount provided by each card in the deck.

However, the nature of the hypothesis is so hard to verify, (and the number of graphs grows so large so quickly) that its verification has only been done up to graphs of 11 vertices, by McKay in 1999.

One would think, that with today's technology and resources, we ought be able to improve upon that bound, and increase N to 12 (or find a counterexample therein). That is hopefully the site of some future work.

### 4.1.1 Parallelizing a Reconstruction Check to 12 vertices

One thought that I had in pursuing this question was: how can we use the massively available scalability of today's cloud based world to extend this problem past 11 vertices? If McKay can do 11 vertices in 1999, one would think that in 2016 we ought be able to do 12 (despite the 160 fold increase in the number of graphs). Here is the algorithm as I would design it.

First, lets define a simple algorithm for checking for equivalent decks. Given a graph $G$, create all of its vertex deleted subgraphs, canonically label and encode each, then append them in lexicographical order. This string is relatively short. For an original graph with 12 vertices, there would be 12 graphs of size 11, if we strip off the first character, which is the same across all graphs of the same size, the encoding is only 10 characters long per graph, meaning that a deck could easily be encoded in only 110 characters (or, 440 bytes). The running time for this algorithm (using NAUTY as a sub-call from a C function (with hardcoded values of 11 and 12 vertices)) takes approximately 2 seconds on average. Thus, if we were able to perfectly parallelize this approach (lets say, across 30 servers), we could

However, we would also need to be smart about when we compare these decks. We cannot use a database to store them (as 440 bytes * 160 billion graphs = 71TB) so we need to generate all graphs which could share a deck at the same time and check at a local level. This is how we would go about doing that:

If two graphs $G$ and $H$ over 12 vertices are going to have isomorphic decks, then there exist 12 graphs of size 11 such that each is a vertex deleted subgraph of both. Given a graph $G$, lets call the seeds of $G$ to be the vertex deleted subgraphs such that the vertex that was deleted is the one with the maximum degree. There can be multiple seeds of $G$ if the graph's maximal degree is repeated multiple times in its degree sequence. It has been shown that across all graph instances, the number of graphs with a maximal degree incidence greater than one is a negligible proportion of all graphs. Thus, our algorithm leverages this fact to do seeded checking for deck isomorphism.

As a side note, it should be obvious that no graphs could violate the reconstruction hypothesis if they have a fully connected node or a fully disconnected node.

Here is the algorithm: a computational node is delivered a graph of size 11, called $S$. If the maximum degree of $S$ is $K$, then the range to search is deemed $[K, 10]$ (as remember, no violating graph can have a fully connected node with degree 11). For each integer $k \in [K, 10]$, we generate $choose(11, k)$ augmented graphs of $S$, for each of the possible connection patterns. Before we move on to the next value of k, we check each of these graphs for deck isomorphism using the technique provided above.

From the small amount of work (unoptimized) that I have done on implementing this checking system so far, it appears that the running time of this algorithm would be about a millisecond per graph of size 11 checked. This would imply that computation of this algorithm over all graphs of size 11 (which would check all graphs of size 12) would take about 12 days to complete, running on the single workstation I was using. If we distribute this workload across many servers, as we could easily do by generating different sets of graphs to check on each (i.e. using NAUTY geng with different values of e), I imagine that this running time could be significantly decreased.

I hope to be able to pursue this verification once my thesis is submitted and I have nothing to do until graduation.

### 4.1.2 Potential Reconstruction Procedure: The Triangle Inequality

One of the first questions I asked when I came across the reconstruction hypothesis was: can Cycles be a part of a solution to this problem? Can we create an algorithm which leverages properties of cycles in order to attempt to reconstruct a graph, or at least place limitations on the possibilities for reconstruction? I puzzled away at this for a while, and came up with an interesting way of incorporating cycles into a reconstruction procedure. I will apologize in advance: this section is particularly notation heavy, and I don't think there is anything that can be done to avoid that.

Lets start with a thought experiment. In a graph $G$, with vertices $x$ and $y$, the number of cycles which pass through a given vertex (lets say vertex $x$) is equal to the number of cycles which pass through the vertex $x$ and the vertex $y$, plus the number of Cycles that pass through the vertex $x$ but not the vertex $y$. This is true for all lengths of cycles we are considering. Thus, we can make a claim of this nature using vectors of length P as our operating unit. We will refer to these vectors by the notation $[x]_G$, denoting the vector of the number of cycles that pass through vertex $x$ within graph $G$, and we will use $[x\bar{y}]_G$ to refer to the number of cycles which pass through vertex $x$ but not through vertex $y$ in graph $G$.

In this notation, the above claim translates to:

$$[x]_G = [xy]_G + [x\bar{y}]_G$$

though this is not revolutionary, when we start to mess around with our graph and its subgraphs, this simple claim can be manipulated to produce much more interesting ones. For example, in the vertex deleted subgraph $G - y$:

$$[x]_{G-y} = [x\bar{y}]_G = [x]_G - [xy]_G$$

We can state the same thing about vertices $y$ and $x$:

$$[y]_{G-x} = [y]_G - [xy]_G$$

If we subtract these two equations together, we get:

$$[y]_{G-x} - [x]_{G-y} = [y]_G - [x]_G$$

This is something we will call the 'tuple' identity. If we have some proposed mapping between the deck cards ($G - y$and$G - x$ in this case) and their proposed associated vertices in their deck cards, we can verify this via the difference between their cycles vector values over the full graph G. A careful reader will now interject: but we don't know the full graph G! If we did, we wouldn't be mucking around with all of this!

We can get around this fact by employing three such 'tuple' identities, to construct a much stronger (and not too information needy) test for possible deck-vertex mappings:

$$[y]_{G-x} - [x]_{G-y} = [y]_G - [x]_G$$

$$[z]_{G-x} - [x]_{G-z} = [z]_G - [x]_G$$

$$[z]_{G-y} - [y]_{G-z} = [z]_G - [y]_G$$

Adding the first and third, while subtracting the second yields us:

$$[z]_{G-y} - [y]_{G-z} + [y]_{G-x} - [x]_{G-y} - [z]_{G-x} - [x]_{G-z} = 0$$

$$([z] - [x])_{G-y} + ([x] - [y])_{G-z} + ([y] - [z])_{G-x} = 0$$

This is the proposed 'triangle equality', which needs to hold for every triplet of vertices within a proposed mapping of vertex to card.

If I had more time (if only, if only), I think I could transform this into a set of boolean predicates which could begin to describe a highly discriminatory set of constraints on the possible reconstructions. The key thing here is that the overall number of triplets for which this can hold is very small, but also sortable. In other words, if you have an idea for a mapping between to vertices and cards, the third could be found quickly by using the remaining constraint of the vector equality dictated by the triangle inequality. Alas, this is as far as I got with this promising use of the Cycles invariant.