NTNU
Innovation and Creativity

Neighborhood Mining in Biological Networks

Kristoffer Stenersen
Sverre Sundsdal

Master of Science in Computer Science
Submission date: June 2006
Supervisor: Magnus Lie Hetland, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Mining biomolecular interaction networks may reveal so far unknown biological properties.

This thesis aims at creating a practical tool for a set of different tasks in one or more networks.
- Comparing proteins based on neighborhood information
- Finding a user specified interaction sub-pattern in an interaction network.
- Finding frequent sub-patterns in one or several networks.

The focus should be on the algorithms solving these problems. This will include finding literature and improving existing work, but also designing new methods when needed.

Assignment given: 20. January 2006
Supervisor: Magnus Lie Hetland, IDI

Abstract

Biologists are constantly looking for new knowledge about biological properties and processes. Bio-molecular interaction networks model dependencies among proteins and the processes they participate. By studying patterns of interaction in these networks, it may be possible to discover implicit information embedded in the network topology. In this thesis we improve existing and develop new methods for investigating similarities between proteins, and for discovering protein interaction sub-patterns.

Cytoscape (Shannon et al., 2003) is a tool for visualization and analysis of interaction networks used by biologists. We have developed an extension to Cytoscape that lets biologists perform the following tasks:

- Compare proteins based on neighborhood information
- Find interaction sub pattern in an interaction network.
- Discover sub patterns in one or several networks.

Our main contributions are improvements to graph mining algorithms gSpan by Yan and Han (2002) and Apriori by Inokuchi et al. (2003) whose original task was the discovering of frequent sub-patterns in a very large set of networks. We have enabled mining a single network and enabled less exact matches.

The graph mining algorithm runs on labeled graphs, and we have used various clustering techniques for this task. The clustering is done through similarity measures between proteins, which we have based on Gene Ontology annotations and experimental data obtained from a ChIP-chip experiment. Our plug-in may easily be extended by adding other cluster techniques or similarity measures.

We verify the results of our implementations and test them for speed. We find that of the two mining algorithms gSpan shows the most promise for mining biological graphs.

Preface

This thesis is a collaboration between Kristoffer Stenersen and Sverre Sundsdal submitted to the Norwegian University of Science and Technology in fulfillment of the requirements for the degree Master of Science. It is written in the period January 16th to June 16th 2006 at the Department of Computer and Information Science.

We would like to thank Assoc. Prof. Magnus Lie Hetland and Prof. Finn Drabløs for invaluable help throughout our project, and Chris Workman, a Cytoscape developer, for feedback on our plug-in.

Working with this project has been very giving. First of all, we have had great times programming our plug-in, and found great value in pair-programming. But also when playing Trackmania and flying our RC planes outside (and over) the department buildings. The constant cheerful atmosphere at Fiol, our computer lab, would not be possible without our good friend Karine Småstuen and fellow students.

Contents

List of Figures

Chapter 1

Introduction

The first section of this chapter give a brief presentation of the problem cases studied in this report and the motivations behind them. Section 1.2 describes our scientific approach to this thesis, while section 1.3 states our objectives. Section 1.4 outlines our main contributions and finally, section 1.5 presents the structure of the rest of this thesis.

1.1 Motivation and problem definition

Recently, large amounts of bio-molecular interaction data have become available. Scientists all over the world contribute their experimental results, building large databases of genetic and physical molecular interactions. Along with the growth of bio-molecular interaction databases, networks of interaction data are created, and the need to explore and analyze these networks emerge. These needs were addressed by Cytoscape (Shannon et al., 2003), an open source software environment for visualization and analysis of biological interaction networks.

This projects aims to implement and adapt methods solving three specific problems. They should be implemented as a plug-in in Cytoscape to enable easy access by a biologist. All problems have some partial overlapping sub problems, and some methods will be reused. Our main input data is biological networks, named graphs in terms of computer science. Here we will give a brief introduction to our set of problems, and our methods for solving them.

1.1.1 Protein finder

In the first problem, a single protein in a graph is given as input, and the task is to find the most similar protein(s) in the same graph. We need to introduce similarity measures based on protein-data external to the graph to be able to compare the similarity between the different proteins. The external data may not give a good classification of a single protein. Under the assumption that a protein's properties are dependent of its interacting neighborhood, we therefore include neighborhoods when comparing proteins. Instead of directly comparing all proteins in the graph to the given protein directly, we thus compare their respective neighborhoods.

1.1.2 Finding network motifs

The second problem is discovering interesting motifs in a single biological network. Given a motif, can a similar pattern be found in a larger graph? A motif is a description of a graph topology or pattern which may or may not include information specific to the nodes. It could simply be a subgraph or it could be some abstract way of representation, like a vector of features.

Known methods solving this particular problem apply to labeled graphs. In a labeled graph, equally labeled nodes are considered equal. Similar to the first problem, external data must be used to be able to measure node similarity. Similarity measures are only one step toward equality, clustering methods can be applied, and nodes occurring in the same clusters are equally labeled.

For example, a large biological interaction network, a chain of proteins and their corresponding links may represent a certain process or a chain of reactions in an organism. For a biologist, it may be interesting to find equal chains in other locations of the graph.

1.1.3 Frequent motif discovery

The third problem is finding patterns across multiple biological interaction networks, but also within a single network. This type of discovery is often solved by data mining which extracts patterns from large datasets. The existing methods apply to labeled graphs, external data and clustering methods must be applied.

For example, the networks may represent interaction networks from different species, and the biologist wants to find similar interactions across the species.

1.2 Our scientific approach

As we consider ourselves computer scientists, not biologists, our implementations and results should be categorized as proof of concept. Our problems are stated by a biologist, and this thesis involves the details concerning the methods we have chosen to apply, how we have improved and adapted them to our test data. The analysis of our work describes the performance and verifies the output of our implementations, but does not discuss the biological usefulness. This is up to the end users of our application. Hopefully, releasing our work as a free, open and easily extendable plug-in to a well known tool for bio-molecular interaction network analysis, our plug-in will prove to be useful by biologists.

1.3 Objectives

Our project aims to implement algorithms solving the described problems, and make them available in a practical tool. Our goal is to customize algorithms to be able to cope with noisy data, make robust implementations, easily applicable for biological scientists. The work of comparing neighborhood information was started by Braute and Rødsjø (2005), however the problems involving finding sub-patterns raises a set of unanswered questions which we aim to investigate:

- Can we use graph mining techniques?
- Does graph mining allow near matches?
- Which graph mining algorithm is best?
- If we can't use graph mining techniques, are there alternatives?

We will describe our methods, adjustments and improvements thoroughly. Also we will test correctness and present performance evaluations of our implementation.

1.4 Contributions

Our main contributions involve a proposed algorithm searching for optimal matches of a given network, in another larger network. Moreover, we have implemented, tested and verified two state of the art graph-mining algorithms. Our implementations are adopted for bio-molecular interaction networks, involving single network mining, clustering for node labeling and various protein similarity measures.

In addition to the underlying implementations, we have released our work as a plug-in to a well known tool for visualization and analysis of bio-molecular interaction data, Cytoscape. The plug-in is easily extended with new similarity measures and clustering methods. Both Cytoscape and our plug-in are freely available on the web. The webpage for the plug-in is `http://www.idi.ntnu.no/~sundsdal/nemi`.

1.5 Structure of the thesis

Chapter 2 presents background theory which is necessary to understand our methods and implementations. The keyword for this chapter is graphs and data mining, but some biological concepts are introduced as well. Chapter 3 presents our methods for the given problems and in chapter 4 we present the implementations and the test results. In chapter 5 we discuss our findings. A summary and our conclusions can be found in chapter 6.

Chapter 2

Background

In this chapter we will include some useful background material which is necessary in order to understand the contexts of the methods presented in the next chapter. Section 2.1 gives formal definitions of graphs, and section 2.2 introduces biological interaction networks. Section 2.3 introduces gene ontologies, and in section 2.4 we talk about Cytoscape, the software environment our plug-in is written for. Before introducing the field of graph mining in section 2.7, we discuss clustering in section 2.5 and graph isomorphism in section 2.6. Section 2.8 reviews some related work.

2.1 Graphs and sub-graphs

Although this project concerns implementations adopted to biological data, we have taken an algorithmic approach and focus on the abstract view. Therefore it is essential to include some background on graph theory. A graph is a pair of sets $G = \{V, E\}$ where V is a set of N nodes and E is the set of edges between the nodes. A node may be labeled, and an edges are either directed or undirected.

A sub-graph $G_s \subset G$ is a subset of nodes $V_s \subset V$ and edges $E_s \subset E$. An induced sub-graph G_i is a sub-graph which consists of a subset of nodes $V_i \subset V$ and all edges $(v_i, v_j) \in E_i \subset E$ if $v_i, v_j \in V_i$ connecting those nodes.

A connected sub-graph is a sub-graph where all nodes are mutually reachable through the edges of the sub-graph. A tree is a connected sub-graph without cycles. Figure 2.1 (a) shows an instance of a graph, while 2.1 (b)-(e) shows examples of the different sub-graph categories.

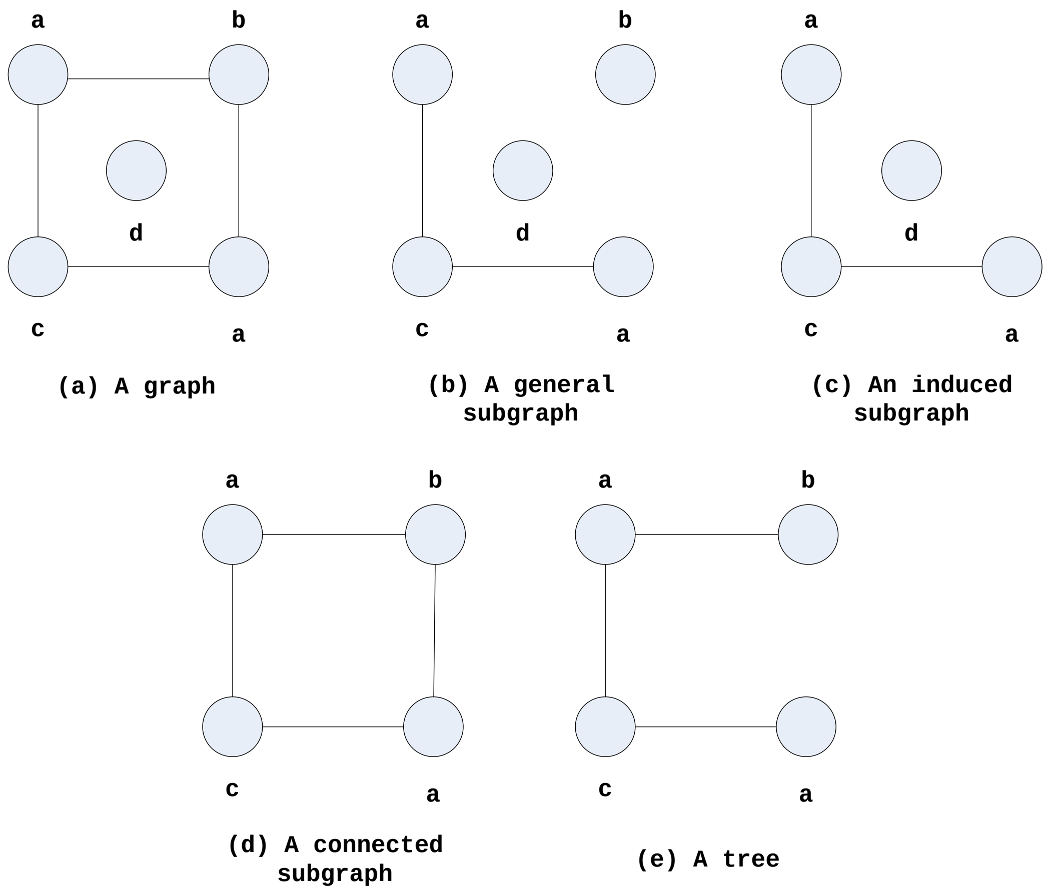


Figure 2.1: A graph (a), and four representative sub-graphs (b) - (e)

2.2 Protein interaction graphs

Molecular biology involves the basic building constituents of species, as well as structural and functional analysis of such. Species are classified into two wide categories, prokaryotic (e.g. bacteria) and eukaryotic (e.g yeast, cat, human). Prokaryotic organism does not keep the genetic material within a nucleus, like the eukaryotic do. DNA (Deoxyribose Nucleic Acid) is stored in the nucleus and contains the genetic material. All the cells in an organism have the same genetic material, even if the cells differ greatly in form and function. An essential part of the DNA are the genes. They encode proteins which are the functional building blocks for cells. Proteins are the principal constituents of cellular material and serve as enzymes, hormones, structural elements, and antibodies. A protein interaction is a sub-process of these functions.

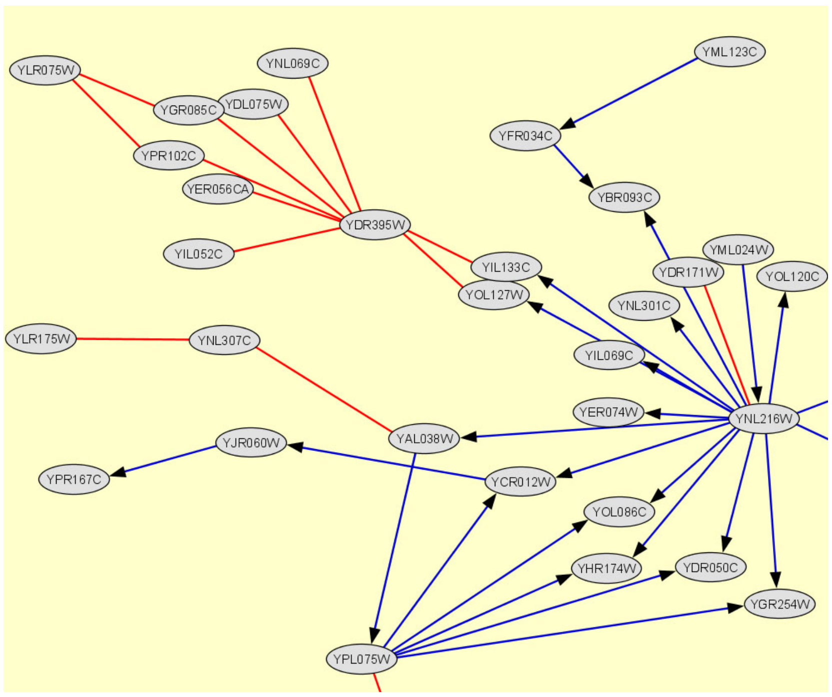


Figure 2.2: Protein interaction graph

Information about protein interactions are collected from published material and stored in public large public databases, for example the Biomolecular Interaction Network Database (Bader et al., 2001). Other interaction networks exists between genes, DNA, RNA, small molecules and complexes, but these are beyond the scope of this thesis. Figure 2.2 is an example graph (red edges are protein – protein interaction, blue with arrow is protein $\rightarrow$ DNA) .

The information stored in such networks is constantly reviewed, corrected and expanded. The information available today may not be the same as tomorrow. Also the information is a simplified view of biology and may not be a correct representation. In short, protein interaction networks are inaccurate and must be handled as such. But still, there is a lot of information to extract from these networks. For example Deng et al. (2004) shows how to use neighborhood information for determining Gene Ontology categories. Sharan (2005) searches for similarities between yeast and bacterial protein networks. The goal of their applied method is to find two sets of proteins, one in yeast and one in bacteria, such that many of the proteins in each set have orthologous counterpart; having the same or similar function.

2.3 Gene Ontology

Gene Ontologies are central to several of our similarity measures presented in chapter 3. When the sequencing of the first genomic sequences became available a surprisingly large fraction of the genes from different species could be considered orthologs, or functionally equal. Information from one species could be used on other species. Knowledge about genes are incomplete and being updated on a daily basis. Some proteins we have very specific information about, others not. An example: One protein is known to be a transmembrane receptor serine/threonine kinase involved in p53-induced apoptosis, the other is only known to be membrane bound. Gene Ontology is designed to handle the problem of varying specificity of information. Updated information can always be downloaded from The Gene Ontology Consortium Website.

The Gene Ontology (Ashburner et al., 2000) consists of two parts. One parts is a controlled vocabulary for describing roles of gene and gene products in cells. The other part is the annotations which maps genes and their products to the gene ontology vocabulary (or categories).

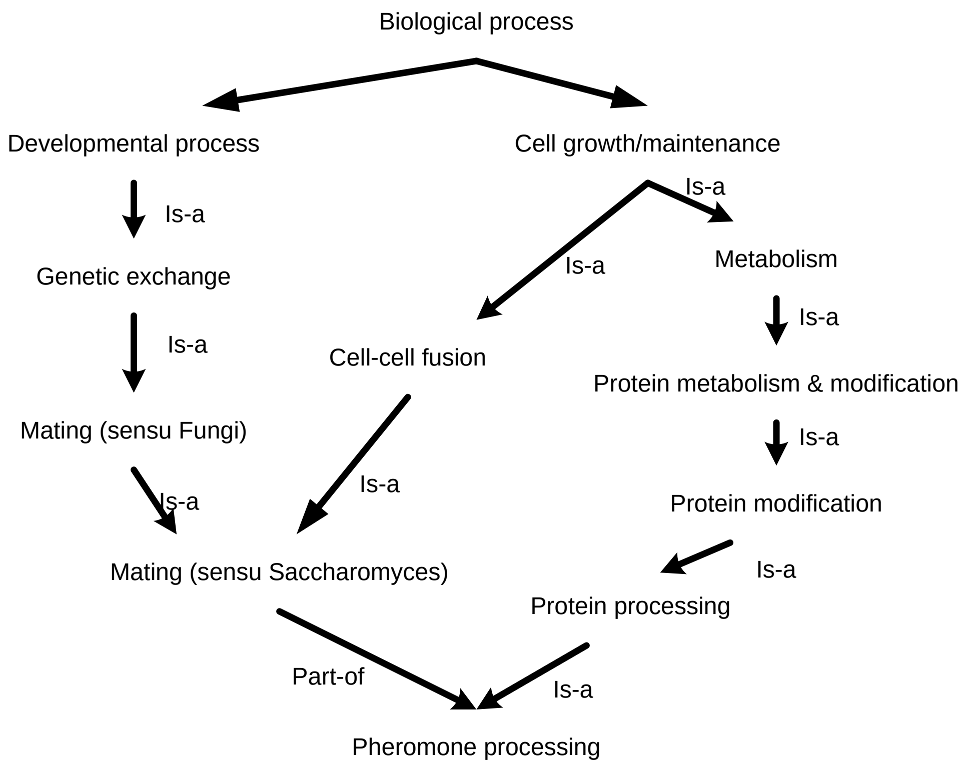


Figure 2.3: Gene Ontology for pheromone processing

The ontology is structured like a directed acyclic graph (DAG). A term can have multiple parents, and it can either be a 'part-of' relationship or a 'is-a' relationship. Figure 2.3 shows an example of the relationships between the parents of *pheromone processing*[1]. The term *pheromone processing* is a step in (part-of relationship) the *process mating* (yeast), an example of (is-a relationship) the process *protein processing*, and an example of the process *protein modification*.

There are three main types of gene ontologies; biological process, molecular function and cellular component. A biological process is a biological object to which the the gene or gene product contributes. Molecular function is a biochemical activity and cellular component refers to where in the cell a gene product is active.

2.4 Cytoscape

The choice of Cytoscape in this thesis is based on a request from Finn Drabløs (Drabløs, 2006). Cytoscape (Shannon et al., 2003) is a software program for visualising and analysis of bio-molecular interaction networks. It allows interaction data, gene expression profiles and other data to be integrated. Other features are available through plug-ins.

Experimental technologies for characterizing molecular interactions, for example microarray experiments, produce considerable amounts of data. Microarray experiments enable scientists to measure the level of transcription activity of every gene withing a cell. Computer-aided tools are crucial for processing and analysis of such data. Various specialized tools already existed before Cytoscape, for clustering, classification and visualization. However, there was still a need for a more general tool able to integrate more model parameters and other biological attributes (Shannon et al., 2003). Cytoscape address these needs in a flexible general-purposed, portable and open source environment for integration of bio-molecular interaction networks and states.

An example plug-in for the Cytoscape environment is BiNGO by Maere et al. (2005). BiNGO, the Biological Networks Gene Ontology tool, is an open-source Java tool to determining which Gene Ontology terms are significantly overrepresented in a set of genes. BiNGO maps frequent functional themes of a given network on the GO hierarchy, and takes advantage of Cytoscape's visualization environment to produce a visual representation of the results.

[1]Example taken from `http://www.yeastgenome.org/`

2.4.1 Cytoscape graph data

Cytoscape stores interaction graphs in two simple ASCII text formats, Simple Interaction File (sif) and Graph Markup Language (gml). The SIF format specifies in the following way:

```
nodeA <relationship type> nodeB
nodeD <relationship type> nodeB nodeC nodeE
```

A node is typically a protein identifier and the relationship can be of the following types:

pp	protein – protein interaction
pd	protein → DNA (e.g. transcription factor binding upstream of a regulating gene.)

These two are the most common but others are also possible:

pr	protein → reaction
rc	reaction → compound
cr	compound → reaction
gl	genetic lethal relationship
pm	protein-metabolite interaction
mp	metabolite-protein interaction

Graph Markup Language (gml) is simply the interaction data plus information about how to view it (layout, colors, edge-ends).

2.4.2 Cytoscape and Gene Ontology data

Cytoscape from version 2.2 handles Gene Ontology natively. It reads gene ontology files in ASCII format formatted like this.

```
0003673 = Gene_Ontology
0003674 = molecular_function  [partof: 0003673 ]
0015643 = anti-toxin [isa: 0003674 ]
0015644 = lipoprotein anti-toxin [isa: 0015643 ]
...
0045174 = dehydroascorbate reductase [isa: 0009491
    0015038 6209 0016672 ]
...
```

This defines an identifier for each term and its relation to a parent. The annotation which describes a gene or gene product is defined as follows:

```
(species=Saccharomyces cerevisiae)
(type=Biological Process)
(curator=GO)
YMR056C = 0006854
YBR085W = 0006854
YJR155W = 0006081
YNL331C = 0006081
...
```

Here all gene or gene products (left) are assigned a gene ontology (right). This example is from species Saccharomyces cerevisiae.

2.5 Clustering

To be able to assign labels to nodes in a network, we apply clustering. The goal of clustering is to simplify the data set. Initially all items in the data may be distinct, then clustering divides data into different groups where all items are considered identical. The goal is that all items in a group are very similar to each other, while items from different groups are dissimilar. Clustering techniques are based on knowledge of the data. Clustering is a form of unsupervised learning.

There are two major types of clustering techniques. There is the bottom up approach called hierarchical or agglomerative clustering which starts of with each element in separate clusters and merge them. While top-down partitional clusterers start with all items in one cluster and try to split them up.

The agglomerative clusterer needs a user specified a distance metric to join iteratively the two nearest clusters until a set number of clusters is reached. The measuring of distance between clusters can be average, max, min or Hausdorf distance between items of the two clusters or some other measure like variance.

A well known a partitioning algorithm is the K-means algorithm which randomly chooses or calculates a set of centers of for the clusters. Then it adds the nearest items and recalculates the centers of the clusters. This continues until some measure of quality is satisfied.

2.6 Graph isomorphism

A common problem when comparing graph information is the so called graph and sub-graph isomorphism problems. Given two graphs $G(V, E)$

and $G'(V', E')$, the sub-graph isomorphism problem is to find the sub-graphs G_s, G'_s and a bijection mapping g between the nodes in E and E' such that the two sub-graphs G_s and G'_s are identical:

$$(v_i, v_j) \in E \Leftrightarrow (v'_i, v'_j) \in E' \mid v'_i = g(v_i) \land v'_j = g(v_j) \tag{2.1}$$

The bijection mapping g is a one-to-one mapping from a set of nodes in G to a set of nodes in G'. All edges connecting these nodes in G must have corresponding edges in G'. The sub-graph isomorphism problem may be extended to sets of graphs by finding a bijection mapping g between the nodes in all sub-graph instances, and their corresponding edges. Also, several sub-graph instances may be found in a single network. Then, g maps nodes within the same network.

The graph isomorphism problem has an unknown computational complexity. It is unknown whether the problem is NP-complete, and all attempts to classify it as polynomial or NP-complete have so far failed. On the other hand, the sub-graph isomorphism problem is known to be NP-complete (Garey and Johnson, 1979). Time requirements of brute force algorithms increase exponentially with the size of the input graphs, restricting the applicability of graph based techniques to problems implying graphs with a small number of nodes and edges.

2.7 Sub-graph data mining

Data mining is analysis technique for extracting patterns from large data sets. These patterns can represent trends in the data that cannot be extracted with ordinary techniques. For example when mining receipt information from a chain of stores one might find that beer and diapers were often bought together, which enables the store to take advantage of that fact.

Many graph mining algorithms take advantage of the frequency anti-monotone principle. If a motif of size k has support s, an extension of the motif to size $k + 1$ will never have more than s support.

Graphs are one of the best studied data structures in computer science. Looking for common patterns in multiple graphs is the typical instance of the sub-graph mining problem. Many ideas and algorithms are adopted from the field of relational data mining, even though state of the art mining techniques do not necessarily perform well on graph based data. Graph based data are by nature more complex, than for example basket case item-sets. The main objective of graph mining is to provide new and efficient algorithms to mine topological substructures in graphs, while the main objective of multi-relational data mining is to mine relational patterns represented by logical languages. The former concerns the geometry and structure, the latter is more logic and relation oriented (Washio and Motoda, 2003).

The earliest studies to find sub-graph patterns from massive graph data were conducted by Cook and Holder (1994). Their approach used *greedy search* to avoid state explosions, and thus did not return the complete set of frequent sub-graphs. They applied a beam search, an optimization of a best-first search. The beam search only unfolds the m most promising nodes at each depth, where m is a fixed number; the "beam with". After this pioneering study, the number of papers released in this field has constantly accelerated.

Another similar approach, developed by the same group, is called *graph based induction* Yoshida K and N (1994). Here similar groups of nodes are replaced by a representative node, compressing groups of nodes into chunks. Chunking can be nested, and the algorithm remembers the link information and can reconstruct the original graph at any time. The chunking is repeated until the size of the graph reaches a local minimum. Sets of equal chunks are returned as sub-graphs.

Kernel function techniques may also be applied to graph mining. The graphs are transformed into feature vectors, a high dimensional feature space characterizing the graphs. Various machine learning, data mining and statistical approaches can be applied if the graph is transformed into a feature vector. The similarity between the returned sub-graphs is defined by the kernel function, and does not necessarily satisfy graph isomorphism.

Apriori-based graph mining techniques are algorithms used in traditional basket analysis. Starting from frequent graphs where each graph is a single vertex, the frequent graphs having larger sizes are searched in bottom up manner by generating candidates having an extra vertex. Apriori applies a breath-first search on the networks. First, all frequent single node networks are found, before expanding to two nodes, etc. Inokuchi et al. (2003) did the initial work on Apriori-based graph mining techniques. Similarly to Apriori, gSpan (Yan and Han, 2002) continuously grows sub-graphs. gSpan performs a depth-first search for sub-graphs, and grows candidate frequent sub-graph edge by edge.

Both Apriori and gSpan are designed to find small sub-graphs in a large set of networks. Bio-molecular interaction networks are often large, and mining such networks may not be done by a direct application of any of the above mentioned methods. Biologists may be looking for sub-graphs in a single network, or a few.

2.8 Related work

Research in experimental biology spans from *in vivo* approaches to *in silico*. *In vivo* involves research on living organisms, while *in silico* is research performed on computers, for example simulations. Our work may be considered a tool for *in silico* research.

2.8.1 Neighborhood mining

Protein neighborhoods was explored last year by Braute and Rødsjø (2005), our thesis continues the work they started. They focused on neighborhood matching, introduced as Protein finder in this report, and concluded that using protein neighborhoods are usable for protein classification. They found that under certain conditions, their proposed neighborhood matching yields a Spearman correlation coefficient of 0.7 with direct function similarity measurements between proteins. We have implemented their protein comparers and their analog to our Protein Finder. Our additions and improvements will be presented in the methods chapter.

2.8.2 Biological sub-graph mining

CODENSE, an algorithm developed by Hu et al. (2005), mines frequent sub-graphs across large numbers of massive graphs. They applied their algorithm to a large set of co-expressed networks derived from microarray experiments, and discovered large number of equally structured sub-graphs across the networks. As opposed to our approaches, their algorithm runs on already labeled networks.

An algorithm by Szpankowski and Grama (2005), called MULE, detects frequently occurring patterns in biological networks. The proposed algorithm is based on existing item-set mining algorithms, performing a depth-first search for connected sub-graphs, expanding the candidate sub-graphs edge by edge. The algorithm involves a pruning heuristic, making it capable of mining large amount of networks in a short amount of time, but is not able to retrieve all sub-graphs found by gSpan. The authors argue that gSpan is not capable of handling very large graphs, and that their algorithm may works as a pruning step of the data, before running gSpan or similar non-heuristic methods.

Chapter 3

Methods

In this chapter we will introduce the algorithms and methods used in this thesis for solving our main three problems. The focus here is to describe how they work and what improvements we have done on an abstract level. First we will discuss how we make protein comparers in section 3.1 and utilize these for clustering in section 3.2. Then in section 3.3 we will describe a method for matching neighborhoods of proteins which is the first of our main problems stated in the introduction. Then in section 3.4 we will describe our straightforward implementation of a basic approach for finding motifs in a network which address the second problem. Finally in sections 3.5 and 3.6 we will describe the two graph mining algorithms we have used to try to answer the last problem, motif discovery.

3.1 Protein-Protein similarity measure

In order to find similarity between sub-networks it is necessary to know the similarity between the nodes. In our project we have used similarity measures that returns a value in the range [0,1]. If two proteins have a similarity of 1.0 they are considered equal. A similarity value of 0.0 means they are dissimilar. This ensures that one measure can easily be replaced by another measure, and we have also the added benefit of generating new composite similarity measures based on a weighted sum. A good similarity measure should be able to return similarity scores for as many protein pairs as possible.

We have created four similarity measures. Three through the Gene Ontology and one using data from a ChIP-chip experiment. Other similarity measures, or comparers, meay easily be added. We have also implemented a composite similarity measure.

3.1.1 Gene Ontology

Most proteins have one or more classifications in the Gene Ontology (GO) system (Ashburner et al., 2000). We use the annotations available from Cytoscape to extract information about common parents. The difference between 'is-a' and 'part-of' relationship is currently not used, after consulting with biologist Finn Drabløs (Drabløs, 2006). The comparers presented in this subsection can be found in the masters thesis by Braute and Rødsjø (2005).

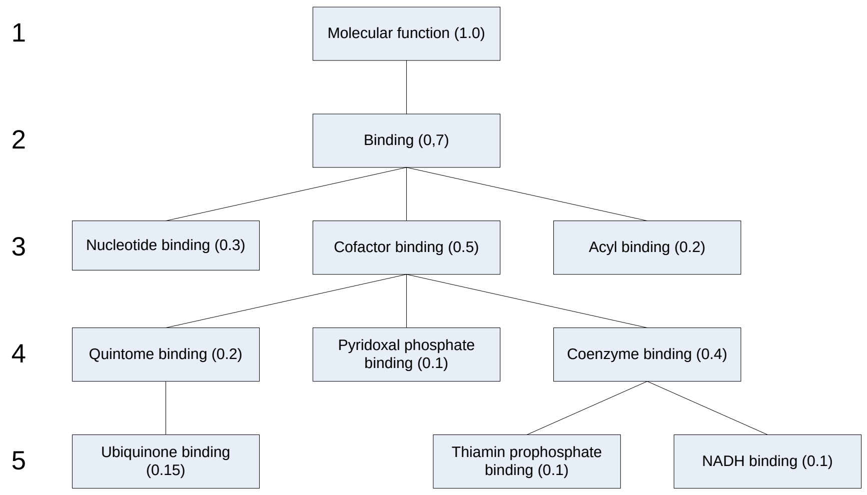


Figure 3.1: Example GeneOntology DAG

Deepest common parent

For each category a protein is annotated there exists one or more path(s) from the root category. These paths are structured like a directed, acyclic graph (DAG). Each step on this path represents an increasingly more specific category. The deeper into the GO a category is, the more information is known about, as deeper GO categories are more specific. A way to utilize this when making a similarity measure is to find all the common categories from these paths for two proteins and extract the deepest one (the category furthest from the root). Call the depth of the deepest common parent category d_{dcp}, and the max depth for all the categories d_{max}. The similarity s_{dcp} is then given by:

$$s_{\text{dcp}} = \frac{d_c}{d_{\text{max}}} \tag{3.1}$$

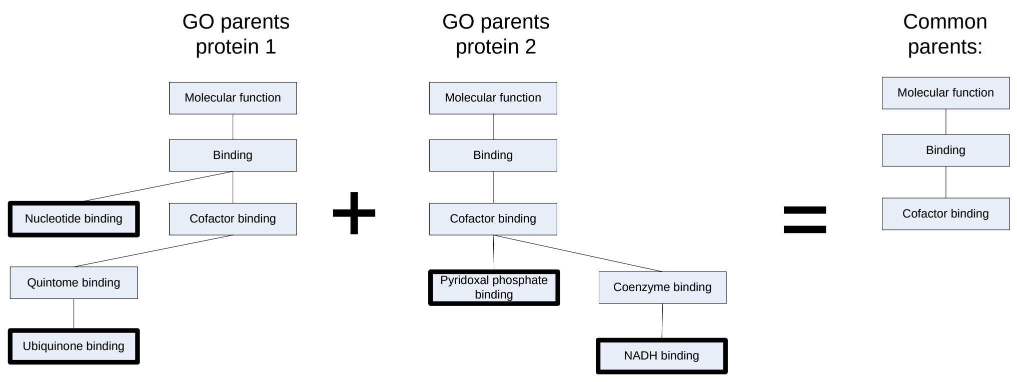


Figure 3.2: Common parents of two proteins

Two proteins are annotated with the example GO DAG in figure 3.1. Protein 1 is classified as Nucleotide binding and Ubiquinone binding. Protein 2 is classified as Pyridoxal phosphate binding and NADH binding. The common parents can be seen in figure 3.2. The score for this similarity will be

$$s_{\text{dcp}} = \frac{3}{5} = 0.6$$

Least likely parent

Instead of looking at the most specific common parent we can look at the most unexpected common parent. The probability that a protein belongs to a category α is defined as

$$P(\alpha) = \frac{\text{number of proteins classified as } \alpha}{\text{total number of proteins}} \tag{3.2}$$

Of all the categories associated with the two proteins, call the probability of the least likely category $P(\text{min parent})$. Call the probability for the common parent with least probability ($P(\text{min common parent})$). The similarity s_{llp} is given by

$$s_{\text{llp}} = \frac{P(\text{min parent})}{P(\text{min common parent})} \tag{3.3}$$

We also made a log-scaled version of s_{llp} to ensure wider spread of similarity measures. We call this "Log least likely parent"

$$s_{\text{lllp}} = \frac{\log P(\text{min common parent})}{\log P(\text{min parent})} \tag{3.4}$$

A high similarity value signifies an unlikely common parent. A very likely common parent like the root node will give the value 0.0. Given the example in figure 3.2 we have the following results:

$$s_{\text{llp}} = \frac{0.1}{0.5} = 0.2$$

and

$$s_{\text{lllp}} = \frac{(\log 0.5)}{(\log 0.1)} = 0.3$$

3.1.2 TF binding

Another approach for measuring protein similarities is using results from a ChIP-chip experiment (Heyer et al., 1999). ChIP-chip experiments are able to estimate the probability that a given transcription factor binds to a given genomic region.

A similarity measure can be created by considering proteins regulated by the same transcription factor(s) more equal than others. By including the probability value, the measures get more graded. A lower p-value indicates a higher probability of interaction.

Example of the input data for this measure:

```
Pid      TFid      Pval
YAL002W  MBP1_YPD  0.0014
YAL002W  PHD1_YPD  0.0085
YAL002W  SIP3_YPD  0.0016
YAL064W  PHD1_YPD  0.00042
YAL064W  SIP3_YPD  0.0031
```

We propose a similarity measure based on the p-values $Pval_i$, $Pval_j$ for the two proteins Pid_a, Pid_b to be compared activated by the same transcription factor(s) $TFid_{a_i} = TFid_{b_j}$:

$$s_{tf}(Pid_a, Pid_b) = \forall TFid \sum \overline{Pval}(Pid_a, TFid) \cdot \overline{Pval}(Pid_b, TFid) \tag{3.5}$$

Where the function $Pval$ returns a p-value if the entry is found for the given Pid and $TFid$, and 1 if not found. Given the example data above, the similarity between the two TFs regulating the genes producing the proteins YAL002W and YAL064W is calculated by $(1-0.0085)\cdot(1-0.00042)+(1-0.0016)\cdot(1-0.0031) = 1.99$. Similarity is normalized against the best match as our clustering methods in the next chapter runs on normalized similarity values. The normalization is calculated by

$$s'_{tf}(Pid_a, Pid_b) = \frac{s_{tf}(Pid_a, Pid_b) - minscore}{maxscore - minscore}$$

where $maxscore$ is the highest score found, and $minscore$ minimum. This ensures that all similarity values are kept in the [0,1] range.

3.1.3 Neighborhood similarity measure

Using one or a combination of the described protein-protein similarity measures described above, we are able to create a similarity measure based on the proteins' neighborhoods. This matching finds the best mutual protein to protein assignment in the two nodes' neighborhoods, and scores the match according to the given single-protein similarity measure.

For example, using GO as a similarity measure when comparing two proteins, the two proteins may only occur in very general GO-categories. General GO-categories are not good classifiers for proteins as they contain too many proteins. But, by also including the two proteins' neighborhoods, more information is added when measuring similarity. Take for example two proteins with a slightly poor similarity, when comparing to one another directly. The two proteins may both only be found in general GO-annotations. But if it shows that the two protein's neighborhoods are very similar, every node of the two node's neighborhoods finds a good match in the opposite neighborhood, it is reasonable to believe that the two proteins could have been given a higher score.

Finding the best node to node matching of two neighborhoods involves the problem of finding the maximum bipartite matching. A more thorough presentation of the problem, and two algorithms solving it, will be presented in section 3.3.

3.2 Clustering

In a protein-protein interaction network all nodes are unique. This means that all sub-graphs occur exactly once in the graph. Applying data mining to such a network will find no new frequent sub-graphs. But if we treat similar proteins as identical we are able to find frequently occurring patterns in the interactions that could be interesting to a biologist.

For some of our algorithms the graph is preprocessed using clustering before starting the data mining. Clustering divides the data into a set of categories in which all elements are considered equal. The common approach is to generate disjoint categories, but in protein interaction networks this is not the best approach. Consider this example, protein A and B share property x which means in some protein interactions A can be exchanged with protein B and the organism would function properly. Protein B and C share property y and can also be interchanged in certain interactions. However in such interactions they cannot be exchanged with protein A. If the clustering puts B in the same category as A but not in the same category as C much information is lost in the interaction network.

A protein therefore should not belong to only one category, but a set of categories. The goal of the clustering should generate categories similar to the the properties mentioned.

3.2.1 Iterative clusterer

The similarity metrics discussed in section 3.1 are non-metric by their nature. The Iterative clusterer handles this very straight forward. The user specifies the tolerance of a cluster, which is the minimum similarity between any two nodes in a cluster.

The iterative clustering algorithm simply starts with a node and adds as many nodes as possible that satisfies the criteria above. Then it finds a node that is in no cluster and repeats the process until all nodes are clustered.

The proteins classified in a single cluster will never be investigated by a graph miner because the support is less than 2. In order to not penalize less studied proteins. Clusters with only one node can be joined to generate a "wild-card" cluster.

3.2.2 Neighborhood similarity clustering

We also propose another version of the iterative clusterer, using the neighborhood similarity measure described in the previous section. When clustering, instead of performing clustering based on direct node similarities, we obtain similarity scores from the nodes' neighborhoods. Again, using neighborhoods for measuring node similarities is motivated by the belief that a protein's neighborhood may give better classifications between single nodes.

3.2.3 TF clusterer

By using the transcription factor regulation data described in section 3.1.2, we are able to create a clusterer independent of similarity measures. A transcription factor regulates a set of proteins with a p-value p. All proteins activated by a transcription factor with p smaller than a preset limit[1] are added to the cluster. In this way the implicit similarity of the TF data is preserved in the cluster.

Clusters that are completely contained within another cluster are removed. Proteins that have no transcription factors associated to it can be added to a single wild-card cluster or separate clusters. This will influence how the algorithms treat proteins with no similarity to any other protein. If a "wild card" cluster is selected, all these proteins will be considered equal by the mining algorithm. If the wild card clustered is not enabled, all these proteins will not contribute to any patterns, as they do not share any similarity with any other node. The wild-card cluster is presented as a check-box in the user-interface.

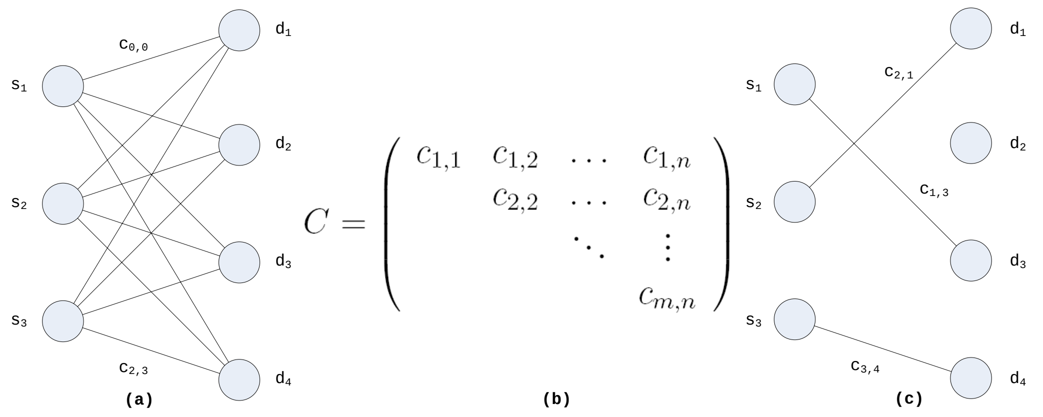
3.3 Neighborhood matching

To be able to measure the similarity of two nodes by their neighborhoods, we need to find an optimal matching of the two nodes' neighbors.

This task can be modeled as the assignment problem, a special case of the transportation problem. The transportation problem has a set of nodes called sources, and a set of nodes called destinations. All arcs go from a source to a destination, and there is a per-unit cost on each arc. Each source has a supply of material, and each destination has a demand. The task is to maximize the sum of flows from sources to the destinations; that is maximizing delivery of material, or meet as much demand as possible at the destination nodes.

In our version of the transportation problem, the assignment problem, there are still two sets of nodes. All arcs go from the source nodes to the destination nodes, but now, every source has a supply of 1, and every destination has a demand of 1 flow unit. In our case, flows are integers, and every supplier will be assigned exactly one destination, and every destination will have one supplier. The solution is enabling the flows giving minimal cost. Figure 3.3 (a) shows an example bipartite matching instance, the cost matrix between the source and the destination nodes (b) and an

[1]The limit is 0.01 in NeMi



s1
s2
s3
d1
d2
d3
d4
c0,0
c2,3
(a)
$C = \begin{pmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,n} \\ & c_{2,2} & \cdots & c_{2,n} \\ & & \ddots & \vdots \\ & & & c_{m,n} \end{pmatrix}$
(b)
s1
s2
s3
d1
d2
d3
d4
c2,1
c1,3
c3,4
(c)

Figure 3.3: An example bipartite matching instance.

optimal/maximal matching (c). Every edge has a weight, a cost, and solving the assignment problem is finding the match with the maximal sum of edge weights or the optimal matching between the two sets of nodes.

Using similarity measures, we are able to assign a protein similarity measure between the two sets of neighborhood nodes. Then, by modeling the source and destination nodes as the two nodes' neighborhoods, finding the optimal match is solving the assignment problem. We implemented and compared two approaches solving this problem; one traditional approach based on Gaussian elimination, and another using a technique of push-relabel techniques. Since this problem is an essential part of the neighborhood mining problem, we will here give a short description to the two methods and make a comparison on realistic data.

3.3.1 The Hungarian Method

This algorithm (Kuhn, 1955) developed by Kuhn was largely based on earlier works of two Hungarian mathematicians, Denes König and Jenö Egerváry. The algorithm performs minimization of the elements in a square cost matrix, and can be easily understood by looking at how it obtain it's final solution: The algorithm performs minimization on the elements of the given cost matrix obtaining a new matrix containing exactly one zero in each row and column. If all other values of this matrix are larger than zero, the zeros then represent the minimal matching of the nodes.

The Hungarian method finds the minimal matching, and we want to find the maximal. Thus, the similarity matrix S is transformed into the cost matrix C by $c_{ij} = max(S) - s_{ij}$. An example follows:

$$S = \begin{pmatrix} .7 & .8 & 1.0 \\ .8 & .5 & .6 \\ .7 & .7 & .9 \end{pmatrix}$$

$max(S)$ evaluates to 1.0 and we obtain the following cost matrix:

$$C = \begin{pmatrix} .3 & .2 & .0 \\ .2 & .5 & .4 \\ .3 & .3 & .1 \end{pmatrix}$$

To find the minimal matrix, the algorithm performs Gaussian elimination to make zeros appear (at least one zero per line and per column). First, the row minimum is found and subtracted from all entries on that row. Then, the column minimum is found and subtracted from all entries on that column. The following illustrates the calculations:

$$C' = \begin{pmatrix} .3-0 & .2-0 & 0-0 \\ .2-.2 & .5-.2 & .4-.2 \\ .3-.1 & .3-.1 & .1-.1 \end{pmatrix} = \begin{pmatrix} .3 & .2 & 0 \\ 0 & .3 & .2 \\ .2 & .2 & 0 \end{pmatrix}$$

$$C'' = \begin{pmatrix} .3-0 & .2-.2 & 0-0 \\ 0-0 & .3-.2 & .2-0 \\ .2-0 & .2-.2 & 0-0 \end{pmatrix} = \begin{pmatrix} .3 & 0 & 0 \\ 0 & .1 & .2 \\ .2 & 0 & 0 \end{pmatrix}$$

Now, lines are drawn across rows and columns in such a way that all zeros are covered and that the minimum number of lines have been used. In this case, there are several solutions, for example one line per row and one per column.

The solution is tested for optimality. If the number of lines just drawn is equal to n (number of rows in the cost matrix), we are done. If the number of lines are smaller than n, a further operation must be done. The smallest entry not covered by the lines is found. This entry's value is subtracted from all other entries in the same row and column, not already covered by lines. Then, a new zero is obtained and one more line is added. Finally, the zeros represents the solution.

In our example, all all zeros are covered by three lines, and the result is returned. There are two possible solutions: $[1, 2, 0]$ (i_0 connected to j_1, i_1 connected to j_2 and i_2 connected to j_0), and $[2, 0, 1]$. Both solutions have a score of 0.5 in the cost matrix C, or 2.5 in the similarity matrix S.

The Hungarian algorithm has a polynomial time bound of $O(n^2 log(n))$ for square matrices.

3.3.2 The Cost Scaling Algorithm

The cost scaling algorithm by Goldberg and Kennedy (1995) extends the minimum-cost flow problem using push-relabel operations and scaling.

The operations are probably best understood in terms of fluid flows. The nodes are now pipe junctions, and the edges are the pipes. Pipes have a flow limit, a maximum capacity between any two nodes *u* and *w*. In addition to the two neighborhood sets of nodes, we also add a source and a drain node. The source node is connected to all nodes in one neighborhood set X and the drain all nodes in the other neighborhood set Y. All pipe junctions have a arbitrarily large reservoir that can accumulate fluid. If a junction has more incoming fluid than outgoing, the overflow is collected in the reservoir. We call the net flow into a vertex v the *excess flow* of v, given by $e(v)$. Nodes with $e(v) > 0$ are defined *active*.

The algorithm is given the costs matrix for the edges C, and the goal is to find the matching giving the minimum cost. Similarly with the Hungarian method, the similarity matrix has to be switched as the CSA finds the minimal matching. Also, the CSA takes integer values only as input. The algorithm's performance depends on the largest absolute matrix value; a higher value should increase the running time as more iterations are expected before termination. Thus, an additional parameter F must be given, denoting the scaling factor of the input data. Given the similarity matrix S containing float values between 0.0 and 1.0, the elements of the cost matrix is calculated by $c_{ij} = \lfloor F - 2F \cdot s_{ij} \rfloor$. For example, given a scale factor F of 500, all values of C will be in the range $[-500, 500]$. Since, in our case, there are only two possible flow values $f(v, w)$ between any two nodes v and w, 1 or 0, the cost of an edge is c_{ij} for an active edge, 0 otherwise.

The cost scaling algorithm works by continuously pushing and relabeling active nodes, explained in the next paragraph. The operations modifies the current flow f and a price function $p : V \to R$ such that f is an ϵ-optimal flow with respect to p. That is, for all flows, $c(v, w) - p(w)$ has to be in the range $[0, \epsilon]$. ϵ is a constant decreasing for every iteration of the algorithm. This gives a successive approximation of a solution, terminating when ϵ is smaller than $1/n$. The algorithm's performance slightly depends on how fast the ϵ is scaled down. Supported by the authors of (Goldberg and Kennedy, 1995), we found that decreasing by dividing by 10 in each iteration gave the best overall results in average for a set of realistic data. ϵ is initially set to F, also as suggested by the authors.

A **push** operation sends a unit of flow from v to w, by increasing $f(v, w)$ and $e(w)$ by one, while decreasing $e(v)$ by one. The **relabel** operation applies to a node v. The operation sets $p(v) = c(v, w) - p(w)$ to the smallest value allowed by the ϵ-optimality constraint. That is, finding the node w that minimizes $c(v, w) + p(v) - p(w)$ in the $[0, \epsilon]$ range. The article also suggest a sequential, modified version of the traditional push and relabel operations, a double-push operation. As their experimental results showed that this version was best in average, we chose to implement this.

The **double-push** operation chooses the two nodes with the smallest and second-smallest reduced costs according to the ϵ-optimality constraint, w and z. Figure 3.4 shows an example of a double-push operation. First, a push operation is applied to (v, w) and $p(v)$ is updated to $p(z) - c(v, z)$. Now, if w becomes active, some other node already had flow to w, another push operation is applied pushing one unit of flow back into that node. Finally, $p(w)$ is updated by $p(w) = p(v) + c(v, w) - \epsilon$. For proofs and details about the double-push operation, please see (Goldberg and Kennedy, 1995).

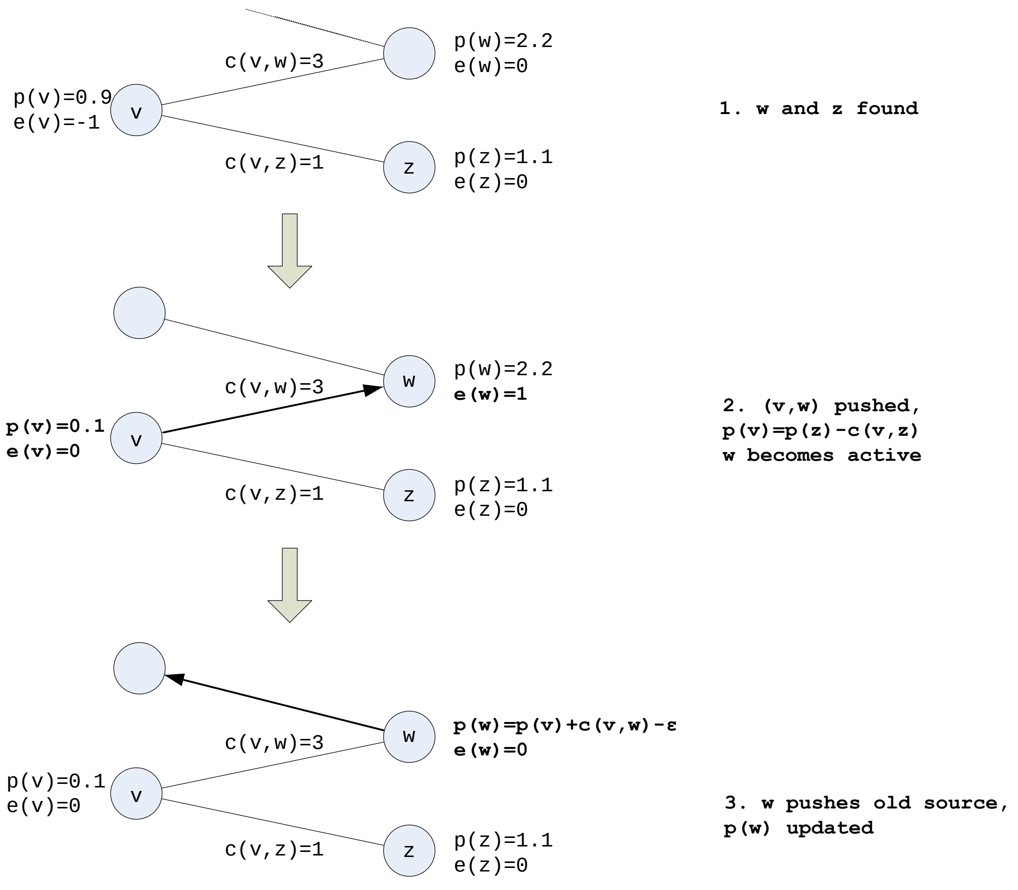


Figure 3.4: A double-push operation.

3.3.3 Scoring bipartite matches

When one of the methods above have found the optimal matching between two node's neighborhoods, a scoring strategy is needed to rank the results. A trivial approach is to sum the scores for each node to node assignment. Using this approach, large neighborhoods are likely to get higher scores. We here propose an approach for weighting assignment scores.

This function uses resampling to score how good a particular assignment is, compared to the score of randomly generated neighborhoods. The source node's neighbors are run against target sets of randomly selected nodes chosen uniformly from the current network. The target sets have equal size to the source's set of neighbors. The weighted assignment score s_w is calculated by

$$s_w = \frac{n_{r>s}}{trials} \tag{3.6}$$

where n_r are the number of random runs scoring equal or higher than the initial assignment score s, and $trials$ are the number of random runs. Different random assignment should be run as many times as possible to give a good estimate of the underlying distribution , but this becomes time consuming for large neighborhoods.

3.4 Motif matching

The second problem we wanted to solve, was the problem of finding the best match of a given graph, a motif M, in another graph, a network N. In this section a the term *motif* is a sub-graph, believed to have matches in the larger network. The following sections describes a straight forward algorithm solving the this specific problem. Even though this implementation is a brute forced attempt including some simple heuristics, this implementation proves to be valuable when comparing to the more complex Apriori and gSpan implementations. Our proposed method does not involve clustering, and it is expected that its output will differ from the other algorithms. Our implementation and results in chapter 4 follows this up.

3.4.1 Problem description

We assume that the number of motif nodes are less than the network nodes $|M| \leq |N|$, which makes our problem similar to the sub-graph isomorphism problem.

Our similarity scores give floating values for the similarities between two nodes. This actually makes our version of the sub-graph isomorphism problem even harder to cope with, as we are not only looking for a similar or equal graph structure of the motif in the network, but we also want to find the best match.

We wanted to implement an algorithm capable of solving problems of any size. Considering the intractability of the sub-graph isomorphism, we needed to sacrifice accuracy to input data size. Also, it is probably not reasonable to require exact motif matches in the network. The creators of the graphs might have missed out some information, and the graphs may lack some nodes or edges. Also, it is reasonable to consider a good node to node match with a small structural difference between the motif and a specific network sub-graph more interesting than a lower node to node match with a perfect structural matching. Our algorithm does not require perfect structural matches.

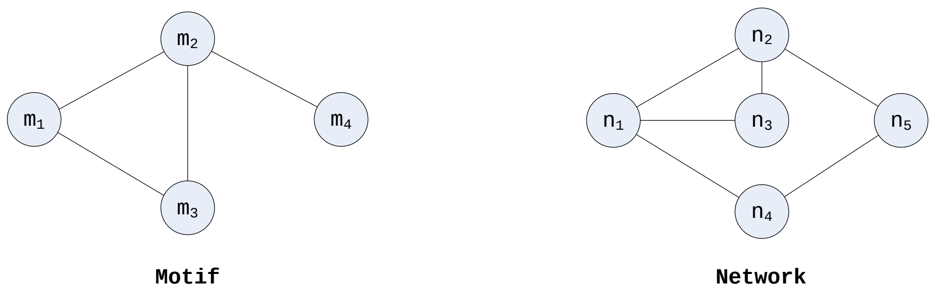
3.4.2 Our algorithm

We hereby propose a depth first search for the described problem. Our approach is based on a complete node to node scan between the motif and the network, where the best match is returned. To make the search tractable for larger motifs, we added a branch and bound factor B to the algorithm. We also allowed for edge inequalities between the motif and network by searching spanning trees.

Figure 3.5 shows an example motif, network and a similarity matrix for every node in both networks. The algorithm starts by selecting one node m_1 from the motif, and one node n_1 from the network. It then searches all matches of m_1 and n_1's neighbors. Note that m_1's neighborhood has to be smaller than n_1's, to be able to continue.

By finding good solutions early, the branch and bounding will be more efficient as more branches can be pruned. Match quality is measured by summing similarity for each m_i node to n_j node match in the neighborhood. For example, a motif neighborhood of two nodes will have a match score of maximum two. Then, another node in the network is tried against m. When all network nodes are tried, m is replaced by a new motif node, and the search continues. The algorithm returns a list the best solutions found.

The algorithm also keeps track of the best score found (*max*) which is used by the branch and bounding. The user can specify the branching factor B. By counting the number of unmatched nodes for a partly assigned



$$Similarity(M,N) = \begin{array}{c} \\ m_1 \\ m_2 \\ m_3 \\ m_4 \end{array} \begin{array}{c} \begin{array}{ccccc} n_1 & n_2 & n_3 & n_4 & n_5 \end{array} \\ \begin{pmatrix} .6 & .4 & .4 & .1 & .3 \\ 0 & .1 & 0 & .7 & .1 \\ .1 & .9 & .1 & .9 & 0 \\ .2 & .2 & 0 & .8 & .8 \end{pmatrix} \end{array}$$

Similarity matrix

Figure 3.5: An example motif, network and similarity matrix.

motif, the maximal possible score value is calculated as the current match score plus the number of reminding nodes. If this score is less than *max* $\cdot B$, the current branch is aborted and the search continues.

Example input data for the Motif Finder algorithm is shown in figure 3.6. Assume the user has set a branching factor to 0.9. The algorithm starts the first branch by matching m_1 and n_1, giving a match score of $.6$ at this point. The search continues matching n_1's neighbors to m_1's neighbors: $\{m_2, m_3\}$ to $\{n_2, n_3, n_4\}$. The first match, m_2 to n_2 and m_3 to n_3 gives a score contribution of $.2$, resulting in a total score of $.8$. The last motif node m_4 can only be matched with n_5, giving a total score of 1.6.

The next branch matches first m_2 to n_4, m_3 to n_2. The branch is not pruned as the highest possible score including the last node is now 3.2 which is larger than *max* $\cdot B$. Finally m_4 is matched to n_5. A new maximum score of 3.0 is recorded. Now consider yet another match, m_2 to n_4 and m_3 to n_3, contributing 0.8 to the total score of 1.4. This branch will now be pruned as $2.4 <$ *max* $\cdot B$, and the search continues.

Note that the edge connecting m_2 and m_3 was not considered in this example, searching for matches with m_1 as the initial node. This is because our approach searches span trees, and does not consider other edges. This makes the algorithm capable of for example finding more motif matches in the network than approaches allowing perfect topological matches only.

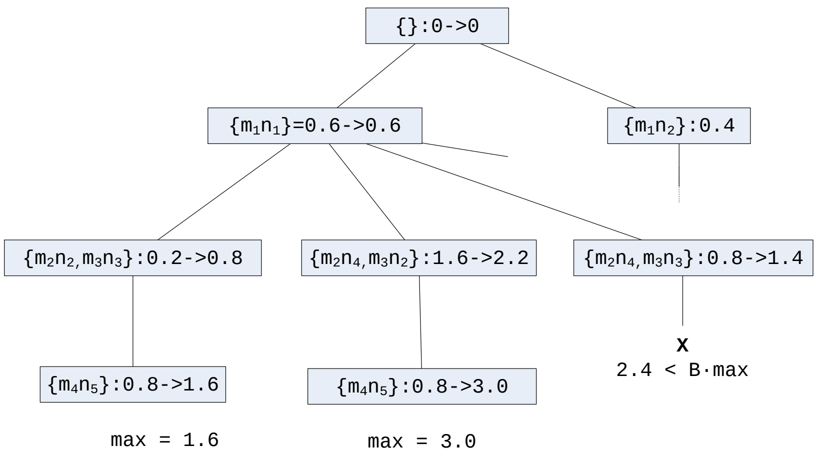


Figure 3.6: Finding the motif in the network.

3.5 Apriori

The problem of finding frequently co-occurring item sets in large data sets is a problem of data mining. One data mining algorithm is called Apriori. Here $k + 1$ item set becomes a candidate frequent item set only if all the k sub item sets are confirmed to be frequent. The infrequent k item sets are pruned from the data set, and then the whole data set is scanned to determine frequent item sets among the candidates.

Using an alteration of the Apriori algorithm as suggested by Inokuchi et al. (2003) we are able to discover frequently occurring sub-graphs. In this section we will describe the method and include any alterations we have done to current published material in graph mining in order to make is more suited to our needs. In the rest of this chapter, a motif is not an actual sub-graph instance, as in the Motif finder. Now, the pattern to search in unknown, and hence the term motif is used about a general sub-graph pattern, believed to occur frequent in a graph.

3.5.1 Representation of a sub-graph

The heart of the data mining process is counting frequent sub-graphs. Counting sub-graphs can be very difficult because there are many ways of representing the same graph topology. In an adjacency matrix for example the rows and columns may be permuted to form a new equivalent representation of the graph.

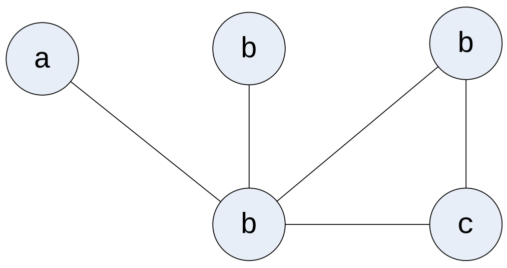


Figure 3.7: Example graph (letters are node labels)

$$X_5 = \begin{array}{c|ccccc} & b & c & b & a & b \\ \hline b & 0 & 1 & 1 & 1 & 1 \\ c & 1 & 0 & 1 & 0 & 0 \\ b & 1 & 1 & 0 & 0 & 0 \\ a & 1 & 0 & 0 & 0 & 0 \\ b & 1 & 0 & 0 & 0 & 0 \end{array}, Y_5 = \begin{array}{c|ccccc} & a & b & b & b & c \\ \hline a & 0 & 1 & 0 & 0 & 0 \\ b & 1 & 0 & 1 & 1 & 1 \\ b & 0 & 1 & 0 & 0 & 1 \\ b & 0 & 1 & 0 & 0 & 0 \\ c & 0 & 1 & 1 & 0 & 0 \end{array}$$

The matrices X_5 and Y_5 both represent the same graph in figure 3.7. But as you can see they have different matrix representation and counting them as different graph structures will be clearly wrong.

To make counting possible, the graph representation for isomorphic sub-graphs need to be equal. Using the approach proposed by Inokuchi et al. (2003) we find a unique representation of a sub-graph. First the nodes of the adjacency matrix will be sorted according to label like in Y_5.

$$M_5 = \begin{array}{c|ccccc} & a & b & b & b & c \\ \hline a & 0 & 0 & 0 & 1 & 0 \\ b & 0 & 0 & 0 & 1 & 0 \\ b & 0 & 0 & 0 & 1 & 1 \\ b & 1 & 1 & 1 & 0 & 1 \\ c & 0 & 0 & 1 & 1 & 0 \end{array}$$

Simply ordering the matrix according to label is clearly not enough. Matrix M_5 represents the same graph as the ordered matrix Y_5. Then we use a special code form for representing the matrix. This code is a simple binary code form that uses less memory. The code is created by scanning the upper triangle elements of the the matrix.

$$X_k = \begin{pmatrix} 0 & x_{1,2}\downarrow & x_{1,3}\mid & \dots & x_{1,k}| \\ x_{2,1} & 0 & x_{2,3}\downarrow & \dots & x_{2,k}| \\ x_{3,1} & x_{3,2} & 0 & \dots & x_{3,k}| \\ \vdots & \vdots & \vdots & \ddots & \vdots\ \downarrow \\ x_{k,1} & x_{k,2} & x_{k,3} & \dots & 0 \end{pmatrix} \tag{3.7}$$

$$\text{code}(X_k) = x_{1,2}x_{1,3}x_{2,3}x_{1,4}\dots x_{k-2,k}x_{k-1,k}$$

For the adjacency matrices Y_5 and M_5 the codes are

$$\text{code}(X_5) = 1010100110$$

$$\text{code}(M_5) = 0001110011$$

The code for M_5 is lexicographically less than Y_5. In order to not check for isomorphism between two graphs we transform the matrix to a least code form, and use this least code form when counting support. The method to find the least code form is defined through the **join** procedure described in subsection 3.5.2.

The code form is concatenated with the corresponding labels of the nodes which gives the *full code form*. The full "canonical code" for graph 3.7 is $abbbc : 0001110011$. "Canonical code" is explained in next subsection.

3.5.2 Joining of sub-graphs

Joining of sub-graphs is used when generating new motifs, finding a least code form and counting frequency. The matrices X_k and Y_k are joined to form $Z_k + 1$ if the following conditions are met.

$$X_k = \begin{pmatrix} X_{k-1} & x_1 \\ x_2^T & 0 \end{pmatrix}, Y_k = \begin{pmatrix} X_{k-1} & y_1 \\ y_2^T & 0 \end{pmatrix},$$

$$Z_{k+1} = \begin{pmatrix} X_{k-1} & x_1 & y_1 \\ x_2^T & 0 & z_{k,k+1} \\ y_2^T & z_{k+1,k} & 0 \end{pmatrix} = \left(\begin{array}{cc|c} \multicolumn{2}{c|}{X_k} & \begin{matrix} y_1 \\ z_{k,k+1} \end{matrix} \\ \hline y_2^T & z_{k+1,k} & 0 \end{array}\right)$$

where X_{k-1} represents the the sub-graph of size $k-1$ and x_i and $y_i (i = 1, 2)$ are the last column vectors of X_k and Y_k respectively. The labels of the nodes needs to be sorted, and labels 0 to $k - 1$ must be equal for matrix X_k and Y_k. Label k (the last label) of X_k must be lexicographically less or equal to label k of Y_k. There are two cases of merging:

- When merging two sub-graphs there is an additional constraint that all nodes must be equal, and the values of $z_{k,k+1}$ and $z_{k+1,k}$ can be found in the complete graph.

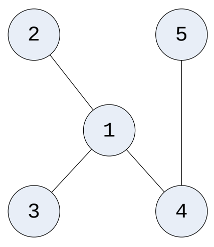
- When generating motifs the value of $z_{k,k+1}$ and $z_{k+1,k}$ correspond to edges not represented by the two matrices and therefore all variations must be generated in order to have all possible motifs.

We call X_k the "first matrix" and Y_k the "second matrix". The matrices X_k and Y_k could easily be switched, but this would produce a different adjacency matrix. The last constraint that must be satisfied before merging is that

$$\text{code(first matrix)} \leq \text{code(the second matrix)}$$

An adjacency matrix constructed by using these rules is called a "normal form" matrix. Any such matrix can be transformed into normal form by reconstructing it. The code for the graph in figure 3.8 (numbers are node identifiers, all nodes same label) is 1101000001, which is not a normal form. Figure 3.9 shows a part of the process to transform this matrix to normal form. Notice that the numbers inside the nodes in the figure are not labels but a unique id for each node. In this example all nodes have same label. It shows how the transform generates normal forms by choosing a least code on each level. Arrow with full line means matrix is chosen as "left matrix" while dotted arrow is "right matrix". Only the first part of the process (for nodes 1 and 2) is shown.

Initially the adjacency matrix is split into sub matrices of size 1. Using the rules above all nodes can be chosen as the "first matrix". On the left side of the figure node 1 is chosen as the first "first matrix" and merged with the rest. This is an arbitrary choice and the right hand side of the



$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 1 & 1 & 0 \\ 2 & 1 & 0 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 & 0 & 0 \\ 4 & 1 & 0 & 0 & 0 & 1 \\ 5 & 0 & 0 & 0 & 1 & 0 \end{array}$$

Figure 3.8: Graph used in normal form example

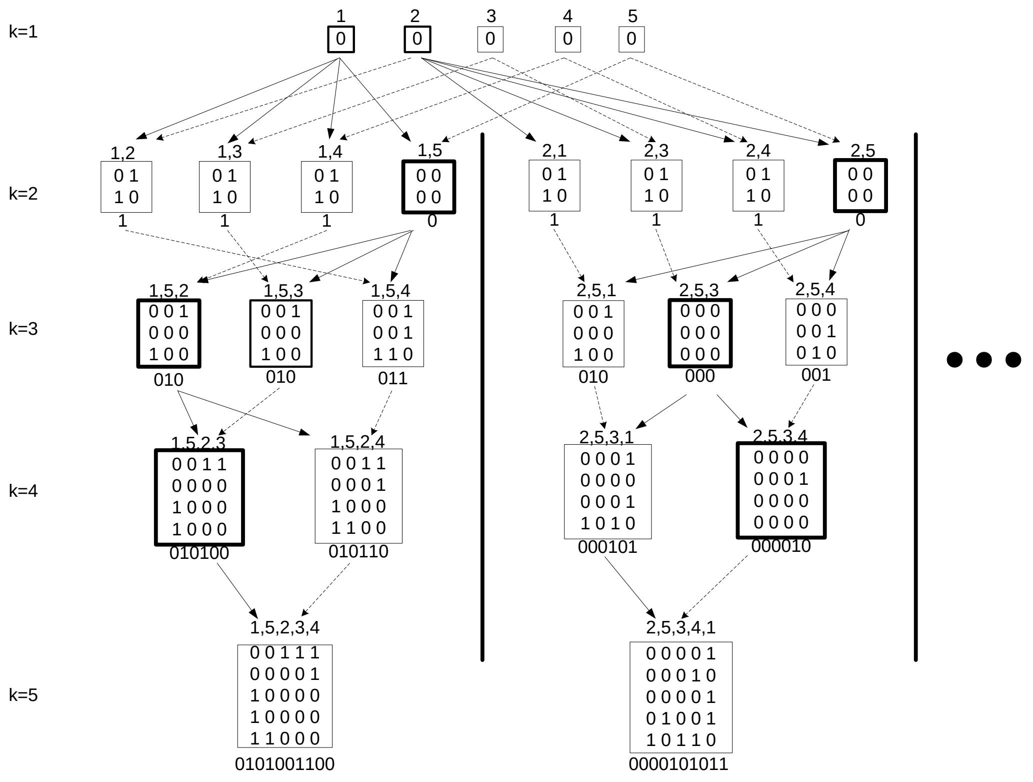


Figure 3.9: Transform into normal form.

figure shows node 2 chosen as the first "first matrix"[2]. Both 0101001100 and 0000101011 are normal forms. But 0000101011 is smaller and indeed even if all possible "first matrices" was completed in this example this would be the smallest. Therefore we call this the *canonical form*.

Finding a least code form is equivalent to the isomorphism problem. Optimizations in the Apriori algorithm allows the use of memoization to speed up this problem. Also when using labeled nodes the number of isomorphisms is greatly reduced.

3.5.3 The algorithm

Figure 3.10 shows how the main part of the Apriori algorithm works. The confirmed frequent motifs of size k are merged to form motifs of size $k+1$. Only motifs where all sub motifs of size k are confirmed frequent is

[2]On the left side on level $k = 3$ two matrices have equal code. Here too, an arbitrary choice was made

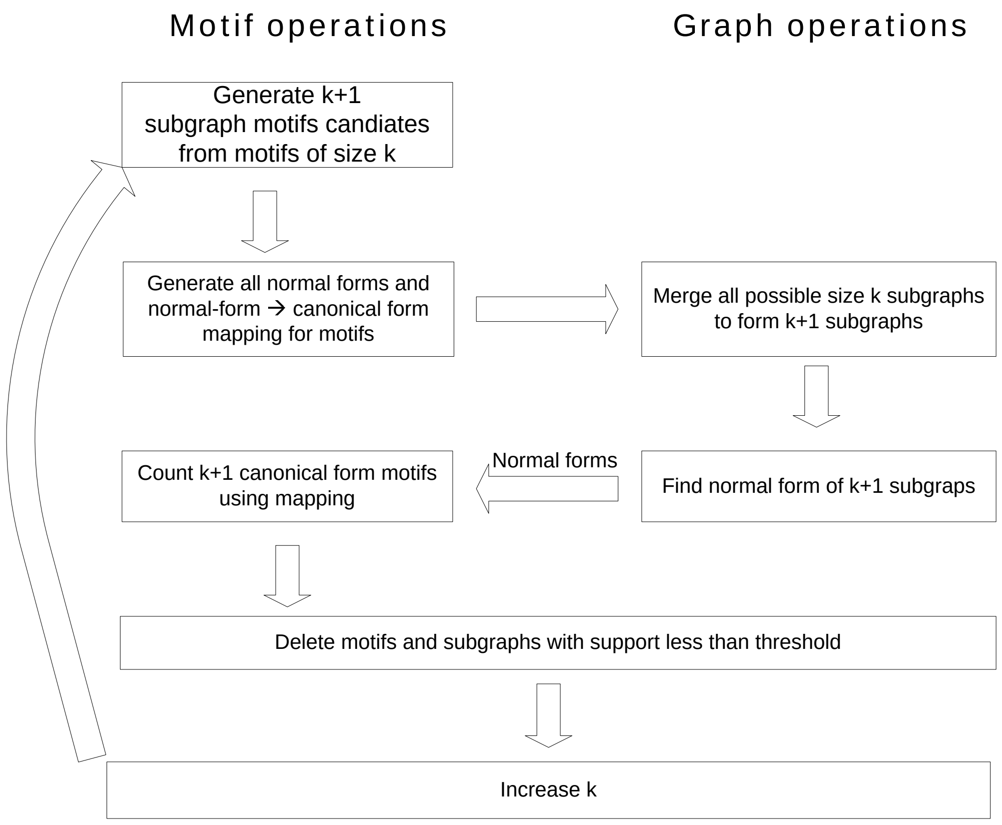

Motif operations
Graph operations
Generate k+1 subgraph motifs candiates from motifs of size k
Generate all normal forms and normal-form → canonical form mapping for motifs
Merge all possible size k subgraphs to form k+1 subgraphs
Find normal form of k+1 subgraps
Normal forms
Count k+1 canonical form motifs using mapping
Delete motifs and subgraphs with support less than threshold
Increase k

Figure 3.10: Flowchart of the main part of the Apriori algorithm

accepted as new candidates. Then all normal forms for these motifs are generated to create a normal form $\Rightarrow$ canonical form mapping.

The frequent sub-graphs of size k are then merged to sub-graphs of size $k+1$. The sub-graphs are sorted lexicographically by labels and code, and then they are merged in the following manner. The sub-graph list is traversed sequentially, the sub-graphs are merged with all the sub-graphs which are lexicographically larger.

The normal form of each $k+1$ sub-graph is found and using the mapping generated previously the canonical form count is updated. All motifs and sub-graphs with *support* $<$ *minsup* are then deleted, k is increased and the process repeats until the desired graph-size is reached. When the mining is done the unconnected sub-graphs are removed.

3.5.4 Single graph mining

The Apriori graph mining algorithm was designed to find frequent sub-graph motifs in a large set (thousands) of small graphs. Our goal was to find frequent sub-graphs in a small set (two or three) of large graphs or just one. With Apriori we decided to focus on finding frequent motifs in one large graph.

In the work by Inokuchi et al. (2003) a graph G_k in the set of graphs G is represented by the adjacency matrix X_k. This matrix is not of normal form and must be normalized. During this normalization all induced sub-graphs of G_k are generated and the normal forms are found and counted.

Generating the normal form of a large graph is not possible because of the time complexity. When working on one graph the switching between "motif" space and "graph space" in figure 3.10 will not result in a pruning of the sub-graph motifs. Instead we generate all possible size 2 induced sub-graphs, add them to a hash map for canonical form $\Rightarrow$ sub-graph instances list. Then we count only *non-overlapping* instances, as discussed below. These are pruned by minimum support and are now the canonical forms with count higher than *minsup* confirmed motifs. Then all sub-graphs of size 2 are merged to size 3 and the process repeats.

In a large single graph many sub-graphs for a given motif will exist. Counting all sub-graphs which map to the same canonical sub-graph motif yields a state explosion very early.

The reason for this is that the frequency anti-monotone described in subsection 2.7 is invalid for single graph mining. To illustrate this consider figure 3.11. We start with a motif like ab:1, two nodes labeled 'a' and 'b' with edge between them. This motif has a support of 2, instances $\{a_1, b_2\}$ and $\{a_4, b_5\}$. Now we add the node 'c' to this motif and get the following motif abc:101 with support 3, instances $\{a_1, b_2, c_3\}, \{a_4, b_5, c_6\}$ and $\{a_4, b_5, c_7\}$, contrary to the frequency anti-monotone.

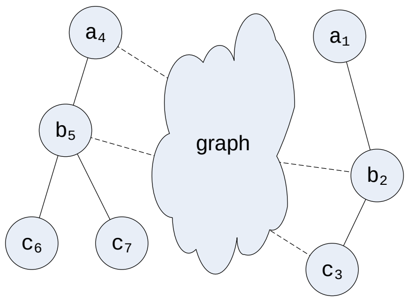


Figure 3.11: Graph example used in frequency anti-monotone example

In reality there are still only 2 “places”[3] in the graph where the motif occurs, not 3. These 3 instances represent two disjoint set of nodes. The counting method should find the maximal number of disjoint sets, or a maximal set packing. Maximum set packing is an NP-complete problem according to Skiena (1998, page. 401). We use a heuristic counting routine which is exact enough for our needs and faster than a more accurate maximal set packing routine.

The counting routine maintains a set of “seen” nodes and for each instance in the graph it counts the seen nodes. If the number of previously seen nodes is less than a set overlap limit, then all nodes from the instance are added to the “seen” set and the counting increments by one.

This counting can result in different values based on which instance it encounters first. This counting is done very often in the graph mining and it is essential that it is efficient. We find this method exact enough for our needs and we will review the effects of this method in section 4.

3.5.5 Nodes with multiple labels

In the complete graph all nodes can have several labels, but in Apriori the nodes can only have one label. To make these representation compatible each node in the complete graph is expanded to several nodes so that each node only has one label. All the edges from the original node are preserved in the expansion.

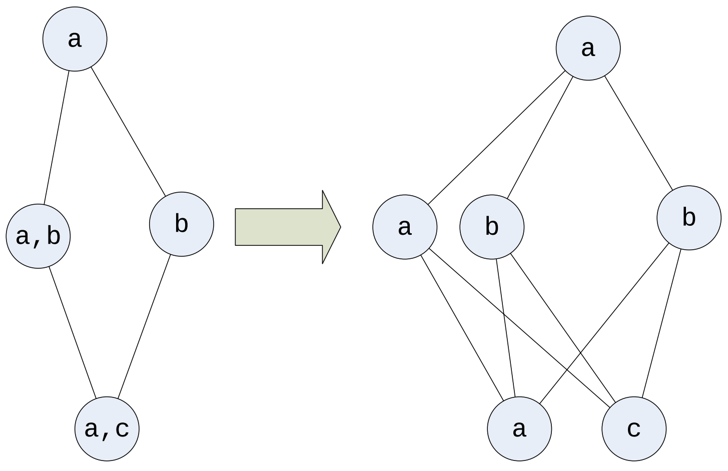
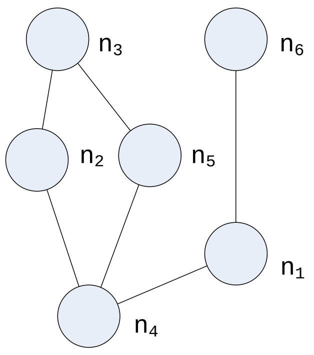


Figure 3.12: Expansion of multiple labeled graph to single labeled graph

[3]Or neighborhoods, not to be confused with the neighborhood matching

The graph is chopped up so that one node is contained in only one sub-graph, then this list of sub-graphs is sorted according to label and node-ID. The candidate motifs are initially all possible labels, with no code (there are no edges yet). The labels are counted and the infrequent labels are removed and the nodes corresponding to that node as well. This is level $k = 1$.

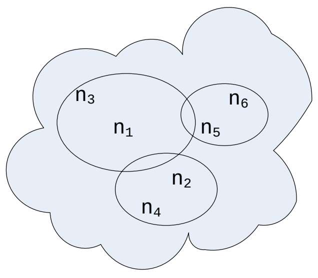
3.5.6 Apriori mining example



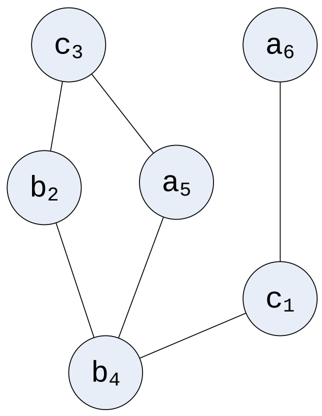
(a) Unlabeled graph

$$\begin{array}{c|ccccc} & n_2 & n_3 & n_4 & n_5 & n_6 \\ \hline n_1 & .1 & .8 & .1 & .3 & .2 \\ n_2 & & .2 & .9 & .3 & .3 \\ n_3 & & & .1 & 0 & .1 \\ n_4 & & & & .2 & .2 \\ n_5 & & & & & .7 \end{array}$$

(b) Similarity matrix



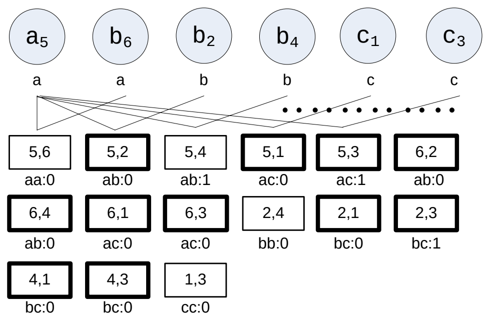
(c) Clustering



(d) Labeled graph

Figure 3.13: Apriori example data

In order to better explain the Apriori algorithm we here include an example. Figures 3.13(a) to 3.13(d) show the transformation of a unlabeled graph into a labeled graph. Through one of our comparers described in section 3.1, a similarity-matrix 3.13(b) is built for all nodes in the given graph. The matrix is reflexive as our comparers do not consider the comparing nodes' mutual ordering. Then, the nodes are clustered using one



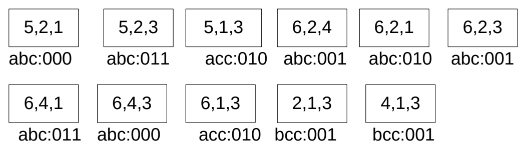
(a) Sub-graphs

aa:0	1
aa:1	0
ab:0	**3**
ab:1	1
ac:0	**2**
ac:1	**2**

bb:0	**0**
bb:1	1
bc:0	**2**
bc:1	**2**
cc:0	1
cc:1	0

(b) Candidate motifs with count

Figure 3.14: Apriori at level k= 2



(a) Sub-graphs

abc:001	**2**
abc:000	**2**
abc:011	**2**
abc:010	**1**
bcc:011	0
bcc:001	**2**

(b) Candidate motifs with count

Figure 3.15: Apriori at level k=3

of the methods described in section 3.2. The cluster id is used for labeling the node and we obtain the labeled graph 3.13(d). In this example, no nodes belong to more than one cluster, and the resulting graph thus have the same number of nodes as the original.

Now, let figure 3.13(d) represent the input data for the Apriori algorithm. This graph contains a frequent motif abc:011, representing sub-graphs $\{5, 3, 2\}$ and $\{6, 1, 4\}$. Here we will show how the Apriori algorithm would find this motif. That is, we are searching for a motif of size 3 and support 2.

Because all motifs of size 1 are frequent, all possible motifs of size 2 are generated, as shown on the left side of the table in figure 3.14(b). The sub-graphs of size 1 start out sorted as shown in figure 3.14(a). Each sub-graph is merged with the succeeding sub-graphs. Then, the counting of the motifs is performed and the count can be seen on the right hand on the table in figure 3.14(b). Frequent motifs and sub-graphs are emphasized.

Merging of all the frequent motifs of size 2 will yield the motifs shown on the left hand side of the table in figure 3.15(b). These are shown in canonical form motifs for illustrative purposes. The sub-graphs of size 2 are merged to size 3, and transformed to normal form shown in figure 3.15(a). Now, the count is computed.

We now have 4 motifs of size 3 that are frequent: abc:001, abc:000, abc:011, bcc:001. Of these only abc:011 is connected, and hence this is the only valid frequent motif.

3.6 gSpan

gSpan, an algorithm proposed by Yan and Han (2002) investigates another approach to sub-graph mining than the Apriori-like algorithms. Apriori has two nontrivial problems: The algorithm suffers from too much memory consumption from candidate set generation. Also, Apriori grows the motifs breadth-first, potentially consuming very large amounts of memory.

gSpan avoids candidate generation and introduces a new canonical graph representation and a lexicographic ordering among sub-graphs. Taking advantage of the lexicographic ordering gSpan also avoids the costly sub-graph isomorphism test. While Apriori needed to transform each sub-graph to canonical form, gSpan ensures that only canonical form motifs are generated. Finally, we modified the gSpan algorithm to better cope with our data sets. gSpan was originally designed to find small sub-graphs in a large set of networks, but we wanted it to find large sub-graphs in one or a few networks. Also, we made a small change in the candidate sub-graph representation.

First, we introduce gSpan's sub-graph representation and the lexicographic ordering among sub-graphs, before describing the algorithm and our modifications.

3.6.1 Overview

The algorithm works similarly to Apriori by first creating small motifs before growing them. Each motif is represented by a tree-like representation, and new edges are only added if the new motif represents a frequent sub-graph in the given graph data set. But instead of merging motifs often representing disconnected sub-graphs, gSpan only grows connected motifs. This restriction reduces the complexity of the problem. A motif may grow into several new motifs, depending on which edge is added. Thus, the search builds a tree of motifs where a node's parent represents a smaller motif (one edge smaller), and a node's children represents larger motifs (one edge larger). gSpan builds this tree, returning the largest motifs representing frequent sub-graphs. The algorithm ensures that motifs representing the same sub-graph never are built twice; an important property concerning search tree size.

3.6.2 Motif representation

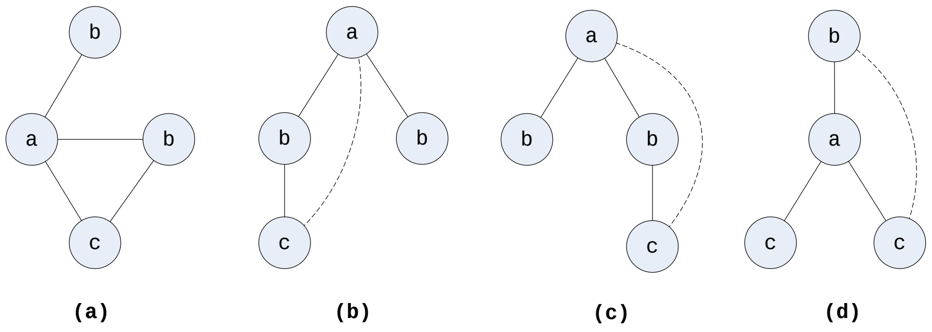


Figure 3.16: A graph (a) and three different motifs (b)-(c) representing (a)

Figure 3.16 shows three motifs (b) - (d), all representing the same graph (a). The motifs consists of two kinds of edges, *forward* edges and *backward* edges. The forward edges are all edges defining the tree structure of a motif, the spanning tree. When growing motifs, every edge connecting a new node to the tree is a forward edge. All other edges, edges that connects between two already discovered nodes, are the backward edges. The *rightmost node* is the latest discovered node, and the *rightmost path* is the straight path from the root to the rightmost node. In figure 3.16 forward edges are filled lines while the backward edges are dotted lines.

3.6.3 DFS code

A *DFS code* is a code representing a motif, i.e (b) - (d) of figure 3.16. In the original article, the DFS code was designed for labeled edges. We made a slight change in this code to match our problem as we did not have any labeled edges. Our code consists of a list of four-tuples, consisting of two integers and two labels (i, j, l_i, l_j). The integers i and j corresponds to the discovery order of the two nodes and l_i and l_j the labels of the nodes when traversing the motif tree. For example, when creating a motif with one edge, the discovery order of the two nodes will always be 0 and 1. Table

3.1 shows the DFS codes for the three motifs of figure 3.16. The article's code also added the edge's label l_{ij}. Our modification does not change the algorithm's correctness nor performance, as the original gSpan algorithm copes with graphs containing only one distinct edge label.

Edge	(b)	(c)	(d)
0	(0,1,a,b)	(0,1,a,b)	(0,1,b,a)
1	(1,2,b,c)	(0,2,a,b)	(1,2,a,c)
2	(2,0,c,a)	(2,3,b,c)	(1,3,a,c)
3	(0,3,a,b)	(3,0,c,a)	(3,0,c,a)

Table 3.1: DFS codes for the motifs (b) - (d)

3.6.4 DFS edge order

To ensure a one to one mapping between the motifs and the DFS codes, an edge sequence ordering is necessary. That is, the mutual arrangement of the DFS code rows. Given two arbitrary DFS edges $e_1 = (i_1, j_1)$ and $e_2 = (i_2, j_2)$, e_1 precedes e_2 if one of the conditions of table 3.2 holds. For forward edges $f(e)$, $i < j$, for backward edges $b(e)$, $j > i$. We name this linear order to $\prec_E$. $\prec_E$ follows the pre-order depth-first search of a motif, where backward edges are searched before the forward edges. The nodes with the smallest labels, sorted alphabetically, are searched first. For example, when creating a DFS code of motif (b) of figure 3.16, the left branch is traversed first. According to the pre-order traversal, edges are emitted to the code as they are visited. If more forward edges were connected to node c, the backward edge (c, a) would be emitted before their traversal. Finally, the search returns to the first node and emits the last edge from node a to b.

	b(e_2)	f(e_2)
b(e_1)	$(i_1 < i_2) \vee (i_1 = i_2) \wedge (j_1 < j_2)$	$(i_1 < j_2)$
f(e_1)	$(j_1 \leq i_1)$	$(j_1 < j_2) \vee (i_1 > i_2) \wedge (j_1 = j_2)$

Table 3.2: Edge ordering $\prec_E$ for DFS edges.

3.6.5 Motif order

As seen in figure 3.16 and table 3.1, several DFS codes may represent the same sub-graph. One of these need to be chosen as the canonical form.

Creating a rule for comparing DFS codes, one is able to choose the smallest as the canonical representation. The following defines the *DFS lexicographical* ordering $\prec_L$ among motifs: $\prec_E$ take the first priority, the node label l_i takes second and l_j the third. Two motifs are compared row by row. If the discovery numbers i and j are equal, the node's labels breaks the tie. For example, comparing three one-edged motifs gives the following ordering: $(0, 1, a, b) \prec_L (1, 2, a, b) \prec_L (1, 2, b, b)$.

Comparing all possible motifs describing the same sub-graph, the smallest, according to the DFS lexicographic ordering defines the canonical motif representation of the sub-graph.

3.6.6 The DFS tree search

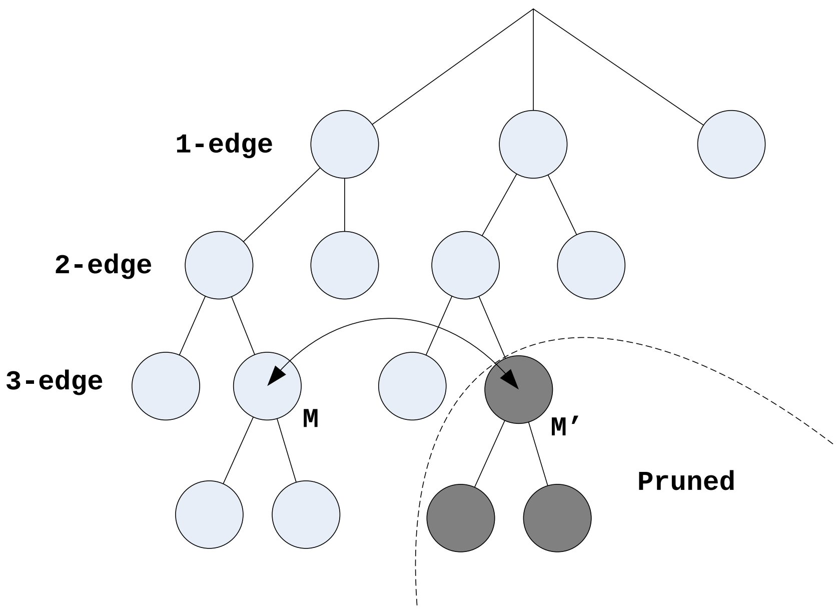
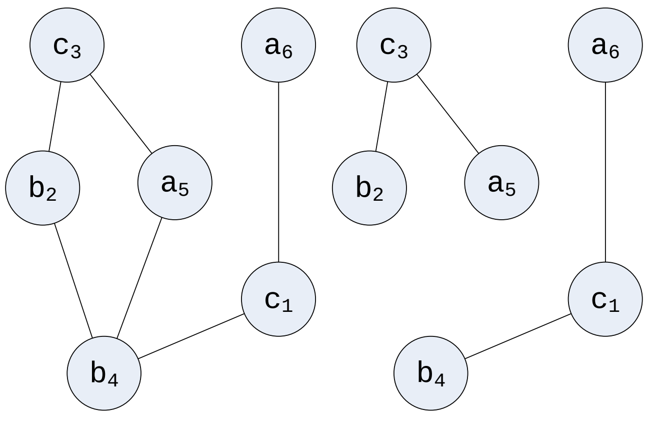


Figure 3.17: gSpan search space and pruning

Figure 3.17 shows an example of a gSpan search tree; a *DFS tree*. Each node represents a motif, and a node's children are motifs grown one edge. The tree is built depth-first, and whenever expanding a node, the lexicographical ordering $\prec_L$ is applied to the node's children. That is, a node's leftmost child (arranged according to $prec_M$) is always the smallest, and is searched first. This ensures that the smallest, and thus canonical, motifs always are visited first.

Whenever a new motif is created and its code is not minimal, we can safely discard the motif and all of its successors. In figure 3.17 M' is pruned as M represents the same motif, but $M \prec_L M'$. Now, discovering two motifs representing the same instances, is the problem of sub-graph isomorphism. In general, this is a computationally hard problem, but gSpan handles this efficiently. The key issue is to ensure that at all times, only minimal motifs are searched. There are two cases concerning edge growth. First, if the first edge of a motif M is e_0, a potential child of M does not contain any edge which is smaller than e_0. If such an edge exists, we know that another motif has been searched before containing this edge, that did eventually grow to be equal to M, only smaller. Second and similarly, whenever growing M with a backward edge into M', if $M' \prec_L M$, M' can be discarded for the exact same reason.

3.6.7 An example



(a) Labeled graph (b) Cleaned graph

Figure 3.18: gSpan example input data

To sum up the algorithm, we will here go through an example. Consider the labeled graph in figure 3.18(a), equal to the example run for Apriori in section 3.5.6.

Let $minsup = 2$, we want all frequent sub-graphs with 2 or more graph occurrences. First, all single edge DFS-motifs are created, and their corresponding instances are recorded. All edges occurring less than $minsup$ times are removed from the network, see figure 3.18(b). In this case, two motifs and four instances are found:
$(0, 1, a, c) \rightarrow \{(a_5, c_3), (a_6, c_1)\}$
$(0, 1, b, c) \rightarrow \{(b_2, c_3), (b_4, c_1)\}$
These nodes define the first level of the DFS search tree of figure 3.19, and

according to the DFS lexicographical ordering, (0,1,a,c) $\prec_L$ (0,1,b,c). Thus the search continues searching motif (0,1,a,c). All instances are searched to find the smallest (using $\prec_E$) frequent additional edge. The first edge found constitutes the next motif at the next level of the DFS search tree. In our case, there's only one edge remaining for the motif, and the motif grows to
$(0, 1, a, c), (1, 2, c, b) \rightarrow \{(a_5, c_3, b_2), (a_6, c_1, b_4)\}$.
Since no more edges can be added to this motif, the sub-graphs instances are saved. Then the search backtracks and finds that no other edges can be added to the previous motif (0,1,a,c) either. Returning to the root of the DFS search three, the initial motif (0,1,b,c) still remains. When trying to extend this motif by the edge (1,2,c,a), the search breaks. As explained in the previous section, since the potential edge (0,1,a,c) would constitute an edge smaller than this motif's first edge (0,1,b,c), and this new edge is a forward edge, we know that the edge (0,1,a,c) has been searched before. The returning list of frequent sub-graphs contains the instances (a_5, c_3, b_2) and (a_6, c_1, b_4).

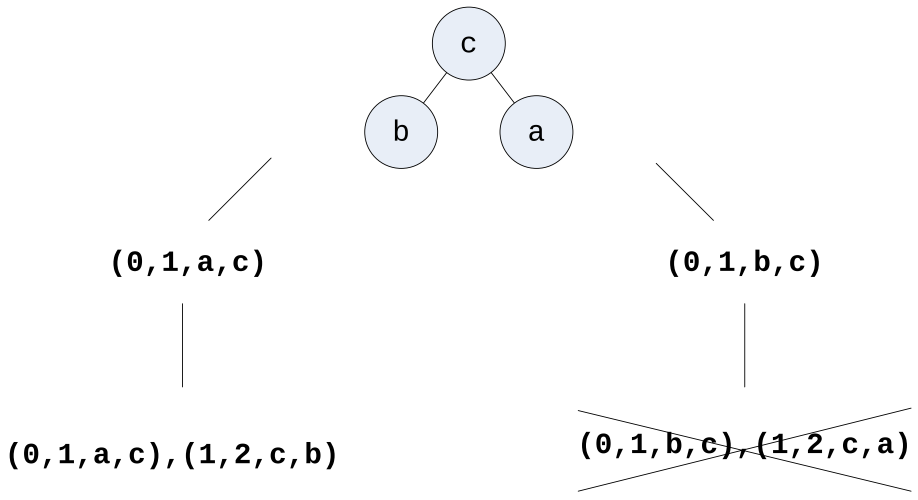


Figure 3.19: gSpan DFS tree search.

Chapter 4

Implementation and results

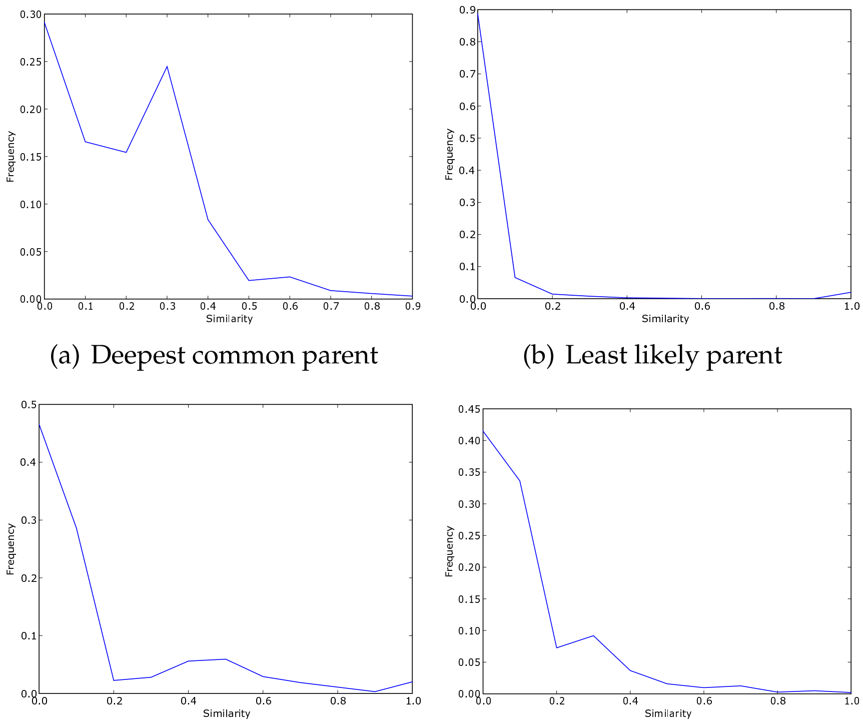
In this chapter we focus on our implementation of the algorithms and their results. The results are evaluated according to speed and correctness. We also describe some implementation specific improvements. An overview of our implementation of the protein comparers can be found in section 4.1 and clustering algorithms in section 4.2. Section 4.3 goes through the neighborhood matching implementations, reviews our improvements and present the results. Sections 4.4 and 4.5 discuss our implementation of the graph miners. Then section 4.6 shows the speed of the motif matcher and mining algorithms. In section 4.7 we test the mining algorithms for correctness. And finally in 4.8 we compare the Motif Finder with the results from gSpan.

4.1 Protein comparers

The focus of this thesis is mining a few or just one single protein interaction graph and making it available in Cytoscape. To do this we needed a way to compare proteins and cluster proteins based on this. Finding out which compare method is best or which clusterers gives the most meaningful results to a biologist is reserved for future work. However we have included some comparers and one clustering method as a proof-of-concept and for our testing.

The protein comparers are inspired by Braute and Rødsjø (2005) and have been used throughout this project, except from the TF comparer. In figure 4.1 we show a statistical plot of similarities when comparing all-to-all nodes in *galFiltered* using Biological Process Gene Ontology. *galFiltered* is a sample network included by default in Cytoscape. *galFiltered* consists of 331 nodes and 326 edges.

For most protein pairs the common parent is less than halfway down the GO DAG, as shown in figure 4.1(a). When looking at the least likely parent, we find that the score declines rapidly and reaches zero at about 20 percent similarity, see figure 4.1(b). More information about this similarity could be extracted as is done with the log factor. Basically it gives a boost to the lower values, but it is essentially the same similarity measure. See figure 4.1(c).



(a) Deepest common parent (b) Least likely parent

(c) Log Least likely parent (d) Composite of the three

Figure 4.1: Frequency plot of similarity scores

These generic comparers might not give a biologist enough freedom so we added the composite comparer shown in 4.1(d). The frequency plot for the Transcription Factor similarity measure is shown in figure 4.2.

A biologist can not only combine different methodologies, but also different GO DAGs in one composite comparer. This allows fine-tuning in Protein Finder, Motif Finder and the graph mining algorithms.

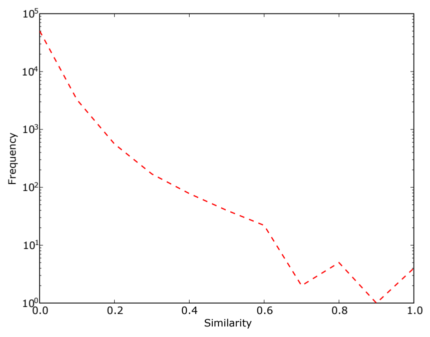


Figure 4.2: Frequency plot for Transcription Factor similarity

4.2 Clustering

The iterative clustering algorithms based on the similarity measures gives us a set of clusters. It's main properties are that all nodes within a cluster are similar to each other within a certain tolerance. The lower similarity limit is set to $1.0 - tolerance$, where *tolerance* is a value between 1.0 and 0.0 given by the user. In addition there are no clusters which completely overlaps another cluster.

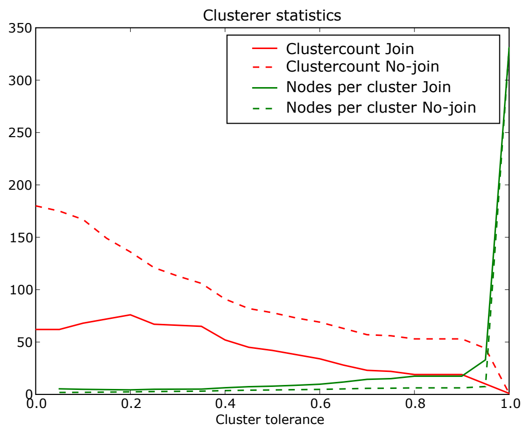


Figure 4.3: Clustering statistics

We noticed that many clusters contained one single protein. That protein was dissimilar (exceeding the tolerance) all clusters. A reason for this could be that there is no available information in the Gene Ontology (or TF database) about that particular protein and hence totally dissimilar to all other proteins. If no information is known about a protein it would be much more useful to consider it a "wild-card". So we decided to add a feature that joined all single clusters. Figure 4.3 shows the impact of this feature. As mentioned in the methods chapter, enabling a "wild-card" cluster will make the mining algorithm consider these nodes equal.

A problem when comparing runs of the graph miners was that the clusters altered between each run. The label of a protein could vary from run to run making comparisons impossible. When building clusters it always started with a random protein which was not currently in any clusters. We fixed this problem by selecting proteins in sorted order based on identifier. It is important to note that this scheme for selecting nodes is arbitrary and has no meaning biologically. After consulting with Finn Drabløs (Drabløs, 2006) we decided to include both approaches for selecting a protein. This was done so that the user can compare results from different clustering runs. Further information of a cluster can be found in the statistics window of NeMi.

The Transcription Factors clusterer however does not have this problem. For each transcription factor we have a cluster. If two transcription factors activate the same proteins the clusterer always creates the largest cluster. On *galFiltered* the Transcription factor clusterer gave 7,46 nodes per cluster on average, and a node was on average in 3,36 clusters. Disabling this reduced it to 5,46 nodes per cluster, but the cluster per node average stayed at 3,26.

Our neighborhood similarity clusterer use the Cost Scaling Algorithm presented in section 3.3.2 for evaluating any two nodes' similarity, based on the two nodes' neighborhoods. The user composes the single node comparer to be used for the bipartite weights, and select the *CSA neighborhood clusterer* in the *clustering method* dropdown. As argued by Braute and Rødsjø (2005), there is a correlation between single node-to-node similarity, and similarity obtained by matching the two nodes' neighbors. This of course depends on the single node-to-node comparer in the neighborhood matching, and they concentrated on GO annotations for their similarity measures. New similarity measures and clusters may easily be added implementing our $ProteinCompare$ interface, as described in appendix C.

4.3 Neighborhood matching

Our Hungarian method (Kuhn, 1955) and CSA Goldberg and Kennedy (1995) implementations were straight forward implementations, even though the article by Goldberg and Kennedy (1995) initially was slightly cryptic and hard to grasp. To ensure that our implementations were correct, we ran series of small test cases on both algorithms and manually verified the results. We also created a Java test class, timing the two imple-

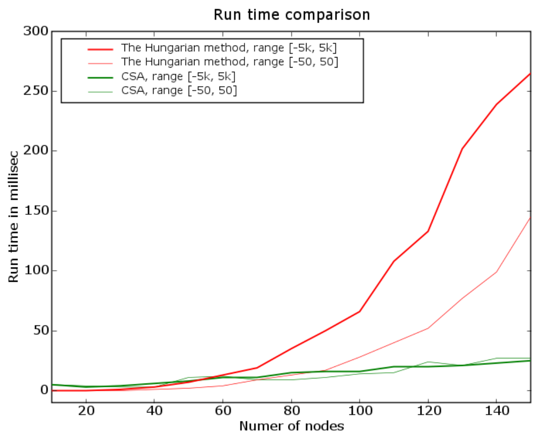


Figure 4.4: A run time comparison of bipartite matching algorithms.

mentations and compared their results. The tester found that both algorithms returned equal results for larger test cases, and the following section presents the timed comparison.

4.3.1 Timing the bipartite matchers

The Hungarian method soon turned out to be too slow for our needs. For smaller neighborhoods, it did not slow down the Protein finder's performance, but for larger networks with larger neighborhoods, and when using the weighting method (3.3.3), the Protein finder became too slow. Figure 4.4 shows a timed comparison between the Cost Scaling Algorithm and the traditional Hungarian method.

The two algorithms ran on a series of randomly generated problem instances. As CSA's performance depends on the input data's range, we randomly chose integers according to the uniform distribution in two different ranges, $[-500, 500]$ and $[-50, 50]$. The number of nodes n are the number of nodes in one of the two equally sized neighborhoods. The number of edges are thus n^2.

CSA outperforms the Hungarian method on both data sets, actually showing less sensitivity to change in the data range, than the Hungarian method. As seen in figure 4.4, the scale factor did not impact CSA's performance in our tests.

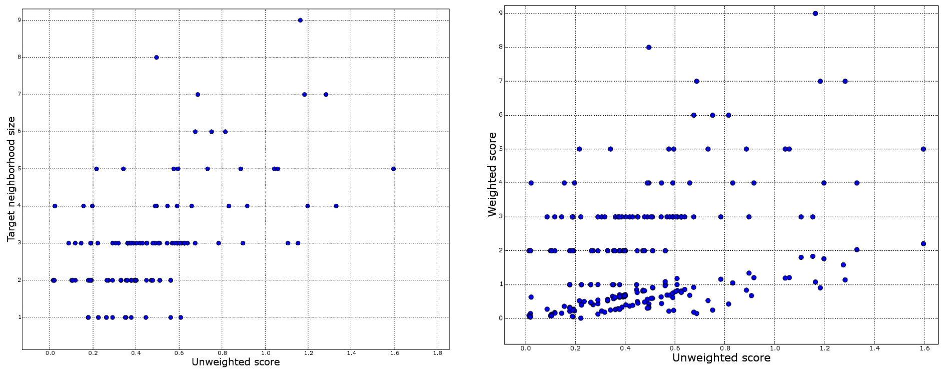
4.3.2 Weighting assignments

As our tests demonstrated, our CSA-implementation performed very well on large networks, returning the best assignments in seconds for a whole network. Also pointed out by Braute and Rødsjø (2005), using the sum of all node-matches to rank the results may not be a reasonable solution. Large neighborhoods get higher scores because they are more likely to contain good matches by chance.

The method presented in section 3.3.3 penalizes large target neighborhoods by comparing the score of an assignment against a large set of random equally sized assignment. The number of random trials should be as high a possible to return an accurate estimate, but as the bipartite matching is a costly operation, the number of runs must thus be chosen carefully in order to keep the method useful.

By manual testing we found that the number of random trials set to $1000/neighborhoodsize$ gave reasonable run times. Figure 4.5(a) and 4.5(b) shows plots from a test run on the $galFiltered$ network. The Protein `YMR043W` was chosen as source node with 18 neighbors. Figure 4.5(a) shows the increasing initial unweighted score increases as the target neighborhoods grow. Figure 4.5(b) shows how the mutual arrangement has changed from the initial unweighted score to the weighted. For example, the two initial highest scoring assignments with a score of 1.6 will now be preceded by many other assignments, ranking from top to bottom.

Randomly created neighborhoods disregards any node dependencies,



(a) Initial assignment score vs target nodes' neighborhood size. (b) Initial assignment score vs weighted assignment score.

Figure 4.5: Weighting assignment scores.

and the function should thus be considered experimental.

4.4 Apriori implementation

Section 3.5 describes the Apriori graph mining algorithm and alterations enabling it to mine frequent sub-graphs in one graph. In this section we will describe difficulties encountered while implementing it and improvements made because of this.

4.4.1 Our first approach

Transforming the code into a code form was easily implemented using a `String` object. We did some performance tests and found that comparing strings was very efficient in Java. Each sub-graph instance was represented by a `SubGraph` class with a a vector of `Node` and `Clusters`. From this we created the code and stored a `stringCode`. This `stringCode` was the concatenation of the `labels` of the node plus the `code`.

The most important part of this algorithm is the **join** operation. Our first implementation of it returned a canonical code, it was very slow so we added the normal form to canonical form map. Initially we counted the frequency of each canonical form code and the frequent sub-graphs continued to the next level, and there were noe check for overlapping nodes.

This implementation didn't perform very well. We could find frequent sub-graphs of size 2, but size 3 took up to an hour for *galFiltered*. Now the process of improving the algorithm started.

4.4.2 Limiting graph size

The example given in subsection 3.5.6 shows that the amount of sub-graphs generated in each level of the search is high. And for bigger graphs it explodes rapidly. In a graph of n nodes the number of sub-graphs of size 2 is $n \cdot (n-1)$ and size 3 will be $n \cdot (n-1) \cdot (n-2)$ in the worst case (if every one of these sub-graphs have enough support). In our case we are looking for a frequent sub-graph which is connected and of max size k. Therefore when counting we can disregard sub-graphs where nodes are further than $k-1$ links away. In the example above, sub-graphs {5,6}, {6,2}, {6,3}, {1,3} can be removed in figure 3.14(a).

This made the algorithm perform much better, but the user often does not know what an appropriate sub-graph limit should be, and overestimation of graph size can still lead to memory overflow and poor performance.

4.4.3 BFS to DFS hybrid.

Still we encountered some memory problems when running the algorithm on a graphs with 100 nodes. Therefore we found another way of counting sub-graphs. The standard algorithm is a breadth first algorithm. This is necesseary to ensure that every $k-1$ sub-motif is confirmed frequent when looking for the k size motif. However, beacause there is no pruning when switching between "motif space" and "graph space" as outlined in section 3.5.4 we can take advantage of that fact,and be able do a depth first search in order to save memory. It will give the same results but use less memory and less time.

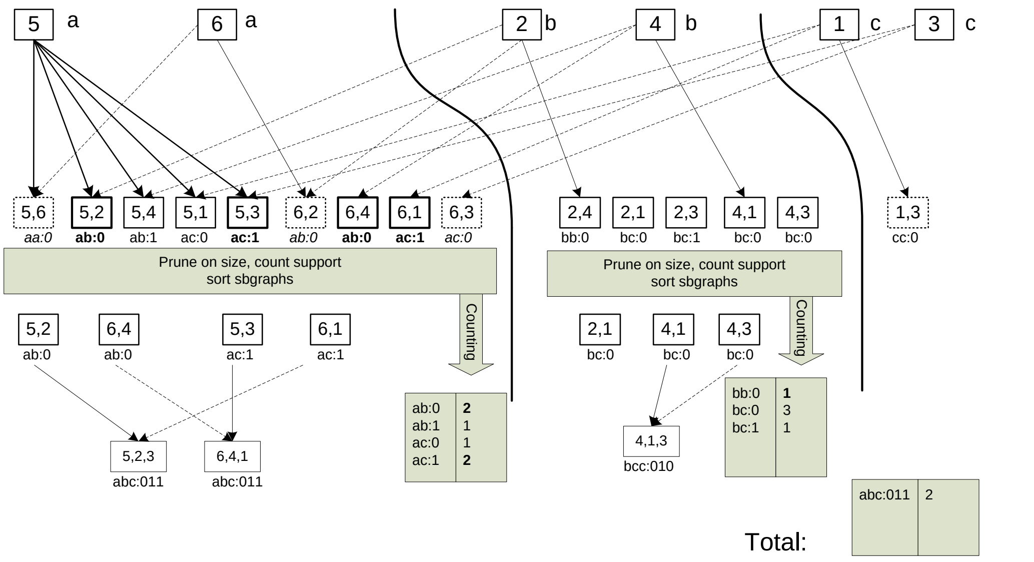


Figure 4.6: Apriori depth first version

Figure 4.6 shows the depth first version of the algorithm. On level $k = 2$ all sub-graphs of size 1 are merged in the standard fashion. The merging starts with the leftmost sub-graph of size $k = 1$ and merges with all sub-graphs to the right to form $k = 2$ sub-graphs. The sub-graphs drawn with dotted borders are pruned because of the size constraint, bold signifies that it is a confirmed frequent motif.

When all sub-graphs with label 'a' have been chosen as first matrix we have generated all $k = 2$ sub-graphs that start with label 'a'. First we prune sub-graphs {5,6}, {6,2}, {6,3}, {1,3} because of the size constraint, then we count for support and remove the infrequent sub-graphs. Then the subgraphs are sorted. We have now reduced the number of sub-graphs starting with ′a′ from 9 to 4. Then we start merging the sub-graphs from the left to the right. When all the sub-graphs starting with ′ab′ are created we stop and repeat. We now count the pattern "abc:011" and save it. We can now safely remove all sub-graphs starting with "ab". Further merging are not possible. We can now remove all sub-graphs of size $k = 2$ with labels starting with "a". This process now repeats with all sub-graphs starting with ′b′ as shown in the figure.

Our speed improvements over the basic algorithm can be seen in subsection 4.6

4.5 gSpan implementation

Apriori graph mining proved to slow and had too high memory consumption, our experiences was useful when exploring another graph mining algorihm, gSpan. For example we learned that limiting the motif size increased the speed of the algorithm. Therefore we added a the possibility to limit how large a motif can grow in gSpan. In addition we ofcourse included the counting method used when mining single graphs. In this section we will describe some implementation details, difficulties encountered and speedups.

4.5.1 Clustering, labeling, cleaning

gSpan runs on labeled networks, and the user specifies one of the clustering methods presented in 3.2 and its parameters. One or more networks are chosen for mining, and after clustering, the chosen network(s) are rebuilt with with labels according to the node's cluster(s). If a node is found in several clusters, the node is duplicated and accordingly labeled. Then all edges are cleaned for support. If an edge has frequency less than $minsup$, the edge is removed from the network(s). Support is the number of occurrences when ran on a single network, and the number of network the edge is in for multiple networks.

4.5.2 Motif and instance representation

A node in the DFS search tree consists of a motif and a set of sub-graph instances. A motif consist of a list of edges, while the instances only has a reference to the last network node, the parent sub-graph instance and its corresponding network. By only saving a reference to the instance's last node and its parent instance, we were able to save considerable memory, but still be able to rebuild the complete list of nodes through the ancestors. Whenever a closed sub-graph instance is found, the sub-graph instance is rebuilt through the ancestors and stored along with the corresponding motif. This way all DFS tree branches can be safely removed from memory when finished traversing them.

4.5.3 Mapping instances to motifs

To be able to quickly determine whether two motifs were equal, we implemented an hash function based on the motif's edge order. For example, when counting sub-graph instances, it is important to quickly be able to retrieve the correct motif and add the instance to its instance collection. Motifs are kept in sets using the motif's hash function as keys. Our hash function builds unique string based on the ordered collection of DFS edges. For example, the two DFS edges $(0, 1, a, b)$ and $(0, 2, b, c)$ is concatenated to the string $"0, 1ab0, 2bc"$. Then, Java's default string hash method is used to return the hash code. This way, we were able to quickly find any sub-graph instance's corresponding motif.

4.5.4 Instance overlapping

As with Apriori, we found during the initial testing of our implementation that very many sub-graph instances returned from the algorithm were uninteresting due to overlaps. For example, consider a 10-edged motif with 5 different instances, where 9 of the edges actually are the same in all 5 instances, only the last edge differ. This made much of the results difficult to interpret.

We included the counting method described in 3.5.4 which only count non-overlapping instances. This made made a considerable speed-up to the algorithm, by bounding branches consisting of less than $minsup$ non-overlapping instances. A parameter $overlapCount$ was added to determine how many overlapping nodes the instances may have to be considered interesting.

Even if only non-overlapping instances are counted during search, gSpan will still return overlapping instances. gSpan runs on exploded networks, where nodes are clustered and duplicated if contained in several clusters. Before gSpan returns its results, we map all gSpan instances in the exploded net back to its corresponding instance in the original network. Again, we check for overlaps, and remove instances which correspond to the same "place" in the original network.

In the 10-edge motif above we chose one instance which will be shown. This makes the data easier to interpret. The choice is random, but we only highlight places in the graph where a motif occurs. When the data is saved to file all instances are included.

4.5.5 Removing backward edges

When mining biological data we are not that interested in exact matches. In our Motif Finder algorithm we only searched for a matching spanning tree not an exact topological match. This was not possible in Apriori because its motif representation was induced graphs, however gSpan can allow this very easily. By only expanding the DFS tree with forward edges we ensure that the motifs corresponds to a tree. The result of this is that gSpan will run faster because a lot of branches are never explored. There will be less results in the result table which makes it easier to read. In addition, we have not lost any of the "places" in the graph which are interesting for further stidy.

4.5.6 The non-trivial $M \neq min(M)$ test

Section 3.6.6 of the methods chapter outlined the tree that the gSpan algorithm searches; all DFS nodes in this DFS tree must be minimal.

When only growing forward edges the heuristic for ensuring minimality is not as effictive. We had to implement a procedure for determination of a motif's minimality. The procedure made a DFS pre-order depth first search on the motif M, retuning the smallest DFS motif $min(M)$ according to $\prec_E$ described in section 3.6.4. If this procedure returns a motif $min(M)$ smaller than M, M can safely be discarded according to the rule of only searching minimal DFS nodes.

4.5.7 Instance comparison

To be able to say something more about the quality of a motif's subgraph instances, we implemented a method for subgraph instance comparison. This method use the same comparer that the clusterer used prior to search, and the calculated instance comparison score will say something about the mutual similarity between the subgraph instances. All subgraph instances in the labeled network are equally sized for a given motif. By comparing all nodes to all other nodes in every position of a DFS-code, every node in the motif get assigned a average score for all instances. The average of these values are shown as *score* in the user interface. In the following section we will make a experimental comparison of gSpan's results.

Figure 4.7 shows a histogram of a gSpan run on the *galFiltered* and *sampleNetwork* networks, cluster limit 0.8 and using the log least likely parent comparer through the GO annotation. As expected, the most frequent motifs found was exact matches, as the *smapleNetwork* actually is a subgraph of the *galFiltered* network. The reason for the high frequency is the overlapping motifs, discussed above.

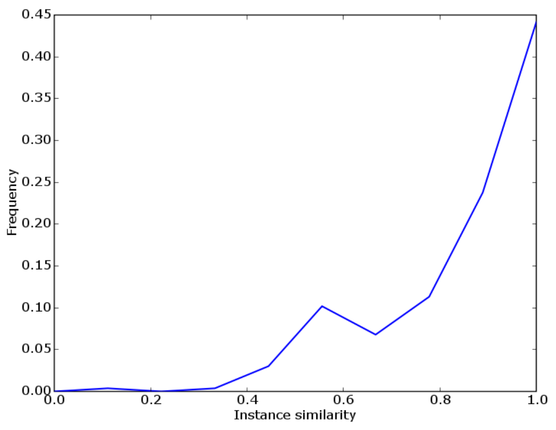


Figure 4.7: gSpan's instance similarity.

This test showed that gSpan's instances actually do have high similarity score, when comparing their mutual nodes using the same comparer as used through clustering. It does not say anything about whether gSpan has left out instances in the original network. This will be investigated in section 4.8.

4.6 Comparing Apriori, gSpan and Motif Finder

To show the actual performance of our methods, we will in this section make a run-time comparison of the Apriori, gSpan and Motif Finder algorithms, ran on the same data sets.

4.6.1 Test setup

The Apriori and gSpan algorithms run on labeled networks, while the Motif finder compares the nodes directly through the chosen comparer. Our Apriori implementations discovers motifs in a single network, gSpan may run on both single and multiple networks, and the Motif Finder runs on exactly two networks. We timed two versions of the Apriori algorithm. Both the original implementation and the improved DFS hybrid approach.

We manually created two data sets. Both sets had a connected *network*, and a connected *motif*. We wanted to compare how the algorithms compared on growing data sets. In every iteration of the test, we grew the motif by one node. The two data sets' networks consisted initially of 13 and 40 nodes, and the motif's two nodes only; both sub-graphs of the galFiltered sample graph. In every iteration of the test, all algorithms were ran on the data set ten times, before the motif was grown with a single node and it's connecting edge.

A log least likely parent comparer (through the molecular function GO) was used for the clustering, resulting in an expanded motif of two labeled nodes and an expanded network of 64 labeled nodes for the largest network, and 20 labeled nodes for the smaller network.

The when doing a multiple graph mining involving a small and a large network it is comparable to what Motif Finder does. Motif Finder and gSpan were run on both motif and network. To make the comparison as fair as possible, the *maximum motif size* was set to the size of the motif graph. Since Apriori only runs on a single graph, and we used the same value for the *maximum motif size* parameter.

4.6.2 A timed comparison

Figure 4.8 and figure 4.9 shows the timings of series of runs on the two data sets. On the larger network, Apriori quickly became too slow because of the exhaustive candidate generation. But on the small networks, it did actually run faster than gSpan. Our improved implementation did make a significant effect, even though the algorithm still was intractable for larger sub-graphs in larger networks. gSpan has a greater start-up cost because it explicitly creates the labeled networks, while Apriori runs directly on the clustered data.

Motif Finder used two seconds on the size 10 motif when ran on the largest network. This algorithm's performance greatly depends on the size of the motif. Our test shows that the run time roughly doubles for every node added to the motif, for this network. On more dense networks, the

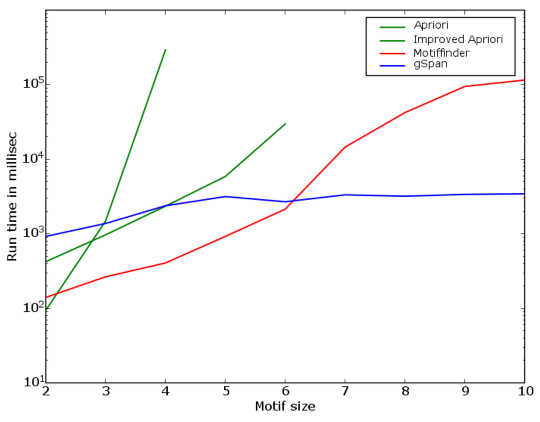


Figure 4.8: A run-time comparison of using network size 40 nodes.

number of possible motif assignments will increase.

gSpan's run-times did not seem to vary much for these input data. In our test, gSpan has advantage to Apriori as it counts support in two networks. gSpan's DFS tree will not grow bigger than the relatively small motif.

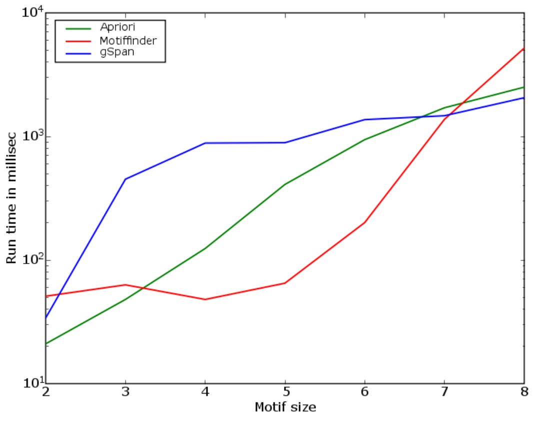


Figure 4.9: A runtime comparison using network size 13 nodes.

4.7 Validating the graph miners

To validate the results our graph mining algorithms we look at two central aspects:

Correctness : Are all the returned sub patterns frequent?.

Completeness : Are there more frequent sub-graphs than the ones returned by the algorithm?

To test for correctness we implemented a verifier in Java that went through the list of sub-patterns and investigated each item in its instance list. An instance list is a set of lists of nodes that the algorithm reports matches the motif. We checked that all the edges in the motifs was present in all node lists, also verifying that the order of the nodes was correct. Both gSpan and Apriori were verified for correctness in several different large and small networks and with different similarity measures and clusterers. They both performed flawless.

The completeness of graph mining run is more difficult. We did this in two parts. First by comparing results of gSpan and Apriori on real graphs, then by inserting some motifs into graphs and testing whether gSpan found them all.

4.7.1 Single network mining comparison

The main problem when comparing the two algorithms is the difference in motif representation. The motifs of Apriori represent induced sub-graphs which are more specific than our gSpan motifs which represent any connected sub-graph. Comparing such motifs would be difficult because we would have to find the possible induced sub-graphs from the gSpan motifs.

In our implementation of gSpan we have removed backward edges which makes the motifs represent a spanning tree. We will therefore have less possible gSpan motifs that represents the spanning tree of an induced Apriori sub-graph. gSpan will have more resulting motifs because of the less specific nature of its motif.

Test setup

We implemented a tool in Python that compares the saved results of gSpan and Apriori. It converts the Apriori motif into one or several minimal DFS codes. Then it compares the instances found by the two algorithms.

We compared equal sized motifs and enforced the same instance overlapping restrictions when counting. All tests were performed on *galFiltered* from the sample data included in Cytoscape. Support was set to 2 and 5 and "join single clusters" turned on. Tests was run with transcription factor clusterer and iterative clusterer with tolerance was set to 0.4 and 0.7 Protein comparers were chosen randomly.

After each run the results was checked with the Python tool and all differences was examined manually.

Results

We started our testing with a simple goal, finding edges in the graph that are frequent. Since the motifs of Apriori and gSpan represent the same sub-graph (one edge) we expect here to find that all motifs match.

Using the transcription factor clusterer the motifs matched for support 2 and support 5. Running the iterative clusterer with support 5 showed a complete match between gSpan and Apriori for both cluster tolerance 0.4 and 0.7. First run with the iterative clustering with support 2 and tolerance set to 0.4 showed a discrepancy between the two algorithms. gSpan found the motif $0, 17C$ not found by Apriori. It's instances were

```
['YLR197W', 'YDL014W'], ['YOR310C', 'YDL014W'],
['YDL014W', 'YOR310C'], ['YLR197W', 'YOR310C'],
['YDL014W', 'YLR197W'], ['YOR310C', 'YLR197W']
```

Looking at the instance set shows that all proteins YOR310C, YDL014W and YLR197W are in both clusters 'C' and '7'. The reason for the discrepancy is the counting method described in section 3.5.4. The post-pruning of gSpan would have removed this. All other motifs matched. For tolerance 0.7 there were 4 motifs in gSpan not in Apriori. All 4 discrepancies can be attributed to the counting method.

Testing with size 3 motifs using gSpan and Apriori we expected to find all motifs from Apriori with gSpan. If we find a fully connected sub-graph motif with Apriori (code form 111), we expect that gSpan will return two minimal DFS codes for that sub-graph excluding one edge. For a co-occurring motif gSpan should find more instances in the graph than Apriori because gSpan uses a more general form of motif.

First we tested with support level 5 using the transcription factor clusterer. This resulted in 2 motifs with support 5 in both Apriori and gSpan. One motif was a connected string of proteins labeled "0". gSpan reported 99 occurrences in the graph and Apriori reported 69. gSpan found all the 69 Apriori found plus 30 others. After examining the graph it was found that all 30 occurrences found by gSpan were connected in circle and hence not found by Apriori.

Using all combinations of clusterers and tolerance as for size 2 we manually checked all differences between Apriori and gSpan. Often gSpan found extra instances, but they were always connected in a circle and would not have been found with Apriori. All other differences in results could be attributed to the differences in counting.

For size 4 motif the same occurred. When testing with support 2, the TF clusterer found 29 common motifs, in 8 of these gSpan found more instances. 14 motifs was only found gSpan and 1 only by Apriori. Using the iterative clusterer set to 0.4 the scores were 20 common, 2 Apriori only, 7 gSpan only. For 3 motifs gSpan found more instances than Apriori. When the iterative clusterer was set to 0.7 the scores were 41 common, 23 gSpan, 1 Apriori. In 10 of the common motifs gSpan found more instances. With support 5 only the iterative clusterer with tolerance 0.7 returned an answer. There was one common motif and one extra found by gSpan.

Discussion

Because of the random element involved in counting both algorithms can report some false positives. This is a false positive in the sense that the instances might not represent truly separate "places" in the graph.

This will be discovered by the human operator and should not be a problem, when the number is so low. In all tests gSpan scored better than Apriori finding more motifs and more instances for each motif.

4.7.2 Multiple network mining

gSpan is designed to mine sub-graphs in multiple networks. Our gSpan implementation supports both single and multiple network mining, and in this section, we will validate gSpan mining multiple networks. By creating a large set of random networks and insert a certain sub-graph into a subset of them, we will be able to verify that gSpan actually find motifs covering at least all inserted sub-graphs. Probably, gSpan will also return other sub-graph instances. These instances' presence will also be verified.

We wrote a tester in Java for the creation of the random networks, and the insertion of a predefined sub-graph. To first ensure that our tester was correct, we ran gSpan on four randomly generated networks with *sampleMotif* randomly inserted in two of them. The test results showed that gSpan did find our inserted motifs among the results, and we manually successfully verified the results.

4.8 Comparing Motif Finder and gSpan

Our own Motif Finder algorithm was initially a brute force attempt for finding a given motif in a larger network. We applied branching, which eventually did make the algorithm efficient even for medium sized motifs. When motifs and networks grow large, Motif Finder will suffer from the explosion of neighborhood permutation possibilities between the motif and the network. gSpan avoids this massive candidate generation and evaluation by only searching minimal motifs according to the lexicographic ordering, but as it only runs on labeled networks, it is reasonable to believe that some matches will be lost.

In this section we will see what the effects of this is by running Motif Finder and gSpan on the same data-set. Section 4.8.1 gives the details on the test setup, and section 4.8.2 reviews the results.

4.8.1 Test setup

We created a small sub-graph of three fully connected nodes, a subset of the $galFiltered$ data-set, containing the nodes (YDR142C, YGL153W, YDR244W). Our test network was also a small sub-graph of $galFiltered$ named $sampleNetwork$, containing the smaller sub-graph. A log least likely parent comparer (through the molecular function GO) was used as comparer, both for gSpan's clustering and Motif Finder's node to node comparer. The cluster limit was set to 0.5, and Motif Finder was run with a branch factor 0 for ensuring all matches found is outputted. After the initial test we ran Motif Finder with a branch factor 0.5 to ensure that it finds the best results when pruning the search space with branch & bound.

4.8.2 Test results

We wrote a small Python script for result analysis. The script was given all instances found by the Motif Finder and gSpan, and returned a list of instances found by gSpan, Motif Finder, and both. As expected, both algorithms found our three-node sub-graph in $sampleNetwork$. Motif Finder gave a score of 3.0 for this assignment; each node returned a full match. The comparer used does not consider nodes' networks, so this only confirms that a node is equal to itself according to the protein comparer.

Both algorithms also found 6 other matches in the network. Motif Finder gave these matches full score, 3.0. This implies that the used comparer gave full score also for these matches, meaning that the iterative clusterer inevitably grouped the assigned pairs of nodes in the same clusters. That is, any pair of nodes considered equal by a comparer will be equally clustered and labeled prior to mining.

gSpan found no other full matches other than those given full score by the Motif Finder, while Motif Finder returned 12 other assignments scored in a range from 1.7 to 2.3. But, gSpan did find 8 smaller versions of these matches; only one node missing. This is because of Motif Finders strength in partial matching and the loss of information during clustering gSpan's data. By running the same test lowering the clustering limit, we found that gSpan did find more full-sized sub-graphs; equal to the sub-graph, but it also found many other matches because of duplicate relabeling. These matches can hardly be considered interesting, as they really just are duplicate node extensions to the original matches (that Motif Finder discovered).

When running the Motif Finder with branching factor 0.5 we verified that it returned all the results with score above $1,5$.

These test showed that two independent algorithms found the correct sub-graph instance in a larger network. The validity of the tests' results were inspected manually, as the motif and network was small enough to be manually verified.

Chapter 5

Discussion & further work

Now we have implemented and tested algorithms for investigating the link structures of protein networks. In this chapter we discuss our work and our contributions to the algorithms. In each discussion section we include suggestions for future improvements. First we talk briefly about the protein similarity measures and clustering in section 5.1. In section 5.2 we discuss our neighborhood matcher. Our improvements to the graph miners and their results are discussed in sections 5.3 and 5.4. Lastly in section 5.5, we briefly discuss the process of developing NeMi.

5.1 Clustering and protein similarity

As noted earlier, the similarity measures and clustering methods are not the focus of this thesis. The similarity measures we have constructed from the Gene Ontolgy are based on the work by Braute and Rødsjø (2005), except from the TF comparer. Many more protein similarities exists and they may be easily integrated into our application. The Transcription Factor data provided to us by Finn Drabløs were easily added both as a similarity measure and clusterer.

The clusters output by the Iterative Clustere,as mentioned in section 4.2, require further study by a biologist. This clusterer is quite time consuming on large networks and we decided not to implement more complex clusterers. However there are a numerous clusterers that are designed to this data type, for example the quality threshold clusterer by Heyer et al. (1999).

When implementing the NeMi plugin we have kept in mind that it should be easy to add new similarity measures and clusterers. Therefore we have included some short descriptions on how to add similarity measures and clusterers in appendix C.

5.2 Neighborhood matching

Our Protein Finder implementation was inspired by Braute and Rødsjø (2005). In addition to implement the CSA algorithm, we also introduced a procedure for weighting an assignment by comparing to a set of random assignments, as large neighborhoods tend to get high scores. In addition to use the neighborhood matching for finding similar proteins to a given protein in the Protein Finder, we also created a clusterer based on the neighborhood matcher. This clusterer takes an arbitrary comparer as input, and use the CSA bipartite matcher on the comparing nodes' neighbors for scoring node similarities through the clustering.

Further work on the neighborhood matching, may be to investigate larger neighborhoods. Our implementation considers neighbors one step away from the source node. By including more steps, the matching will become more computationally expensive.

5.3 Apriori

Adapting Apriori to biological graphs served as an introduction to the field of graph mining and near matching. In subsections 3.5 and 4.4 we introduced some essential features to the algorithm to increase speed and adapt it to the biological graphs.

The speedups added to Apriori allowed us to mine for slightly bigger motifs. On a network with 40 nodes we increased the motif size from 4 to 6. On larger networks like *galFiltered* the original algorithm ran out of memory when looking for size 3 motifs. We have now confirmed that the improved algorithm works for up to size 4 motifs on *galFiltered*.

Near matching was achieved with the loose clustering methods. But Apriori is designed to look for *induced* subgraphs. Tweaking the implementation of the algorithm may give speed increases and memory reduction. This is a fundamental feature of the algorithm and together it persuaded us to explore for a new approach to graph mining, namely gSpan.

However, there are some improvements that can be still possible with Apirori. Work could be done on the motifs to reduce the problem of *induced* graph like introducing a "2" or "3" in the code to signify distance 2 or 3 between the nodes. The counting procedure could be changed to a better approximation algorithm for the maximum set packing. Set packing is NP-hard and even an approximation algorithm will be time consuming since it is performed often. Since we often are interested in finding completely separate subgraph motifs, we don't consider this a problem.

5.4 gSpan

gSpan was described in section 3.6 and some implmentation details was discussed in section 4.5. Here we will discuss our experiences with gSpan. gSpan's canonical motif representation ensures search efficiency by enumerating the search space in a depth-first manner. The original algorithm is designed for mining large sets of smaller graphs, and we have made additions making the algorithm able to cope with larger graphs. Also, we have adapted the algorithm to also be able to search single graphs only.

5.4.1 Improvements

The preprocessing of the input data was similar to Apriori when preparing unlabeled networks to the gSpan algorithm. gSpans user interface lets the user decide which comparer and which clusterer technique should be used as well as the different parameters tuning the search. These parameters makes our gSpan implementation able to cope with larger networks. Under the final testing of our implementation, we did experience heap space overflow in large graphs. An example of such is $yeastHighQuality$, a default sample network in the Cytoscape environment. This network contains 3025 nodes and 6886 edges. When motifs grow large, the active DFS tree branch will become too deep. Similar to the Apriori algorithm, an iterative implementation could solve the heap space problem. Also, increasing Java's heap space by adding the parameter `-Xmx512M` or `-Xmx1024M` when invoking the virtual machine will make the algorithm cope with larger graphs.

Still, we have implemented routines and parameters restricting and making the search more efficient. Tuning these parameters makes Java's virtual machine's default heap space configuration sufficient in most cases.

First, as discussed in the previous sections, adjusting the clustering tolerance will affect the graph miners' performance. A low tolerance will produce small clusters and few node duplications when labeling the nodes prior to mining. gSpan does not consider any edges or node-pairs with lower support than $minsup$, and by creating small clusters, many edges will be pruned away prior to mining. These nodes are the least similar nodes to all others in the network, and adjusting the cluster tolerance is thus a pre-pruning using similarity threshold through the given comparer.

Second, another important feature concerning pre-pruning of the search space is joining single clusters. Single clusters are nodes considered dissimilar all other nodes in the network by the comparer. If the user chooses to disable this option, all these nodes and their belonging edges will be pruned as they does not satisfy $minsup$.

Third, we added a *max size* parameter to the user interface. This parameter defines the maximum motif size during the search. If the user knows he is looking for frequent subgraphs not larger than 15 nodes, he may use this parameter to speed up the search and avoid the mentioned heap space problems.

Fourth, yet another important addition to the gSpan algorithm is our overlap implementation, described in section 4.5.4. The user may choose how many subgraph instance overlaps that should be tolerated during the search. By allowing zero overlaps, the search will be able to prune all motifs representing subgraphs containing any overlaps. The $minsup$ or minimum support parameter is an important parameter. At high $minsup$ values, gSpan will return smaller sub-graphs than using a low $minsup$ value.

Fifth, our mutual instance comparison from section 4.5.7 makes a comparison of all subgraph instances for a motif. This enables the user to reorder the gSpan's result. By default, results are ordered by motif size, but this ordering is based on the degree of similarity between the subgraph instances found.

5.4.2 Overlapping motifs

There is an important difference between overlapping subgraph instances and overlapping motifs. Our gSpan implementation does not return overlapping instances if not decided by the user, but the algorithm returns overlapping motifs.

It is harder to find a way to remove motif overlaps; the problem is to decide which to remove. One approach could be to find a set of overlapping motifs; and only keep the motif(s) representing most subgraph instances. Another could be to keep the largest motif(s), counting nodes or edges. The problem with such approaches is that all resulting motifs may overlap some other motif. If one decides to return, say, the best motif only according to one of the mentioned measures, the algorithm would only return one single motif. As our graphical user interface makes browsing through gSpan's results quite efficient; we did not investigate the problem of overlapping motifs any further.

Removing overlapping motifs at runtime may also give a potential speedup to graph miners at runtime. Again, the problem is to decide which motifs to discard according to the arguments above. Also, one does not know the potential of a growing motif until the algorithm completes the DFS search on this motif.

5.4.3 Instance similarity

The crux of subgraph mining is the complex topological structure of graphs. In addition, all nodes of a biomolecular interaction graphs are unique; only through our supplemental similarity measures are we able to give similarity score between the mutual network nodes. We have looked into two distinct approaches solving the problem of subgraph mining. The Motif finder, comparing nodes to each other directly, and the graph miner, running on clustered graphs.

In section 4.8 we made a experimental comparison of Motif finder and gSpan's output, searching for a small subgraph in another, larger graph. We have argued that clustering does affect the level of subgraph instance looseness, in terms of exact matching. Our tests also confirmed this. Motif finder scores the matches in floating values, being able to rank small but similar matches higher than larger matches, with poor scores. Our graph miners returns motifs sorted on descending size; while our Motif finder use the sum of node-to-node score for all nodes in the match. Therefore, we have implemented a method for comparing a motif's instances to each other. This enables the user to sort the results in a different manner, not only by subgraph size (default), but also by subgraph instance similarity.

5.4.4 Near matches

Inspired by our own Motif finder, we wanted to relax gSpan's exact subgraph matching. Biological data may be noisy and incomplete, and some level of incomplete matches between the instances should be tolerated. Our modification to gSpan was to only search forward edges, according to the DFS codes presented in section 3.6.3. In effect, subgraph instances now may not have all edges present, between the nodes. It is only required that the set of nodes is connected as a tree.

Furthermore, clustering partially influences the non-exact matching issue. Our cluster limit parameter, a slider in gSpan's user interface, decides the width of a cluster. If the user decides to set a low cluster limit, only nodes with high similarity according to the chosen similarity measure will be put in the same clusters. This will again generate few node duplicates, and the miner will find fewer, but more specific motifs. On the other hand, if the cluster limit is set high, many nodes will occur in several clusters, get several labels and thus duplicated. The miner will more easily find more and larger motifs. This way, the level of matching tolerance may be adjusted through the user interface.

5.4.5 Further work

As the number of tuning parameters grow, the usability of a tool is harder to ensure. A future improvement to our gSpan implementation may be to include a analysis of the input data, to be able to propose a set of parameters making the search tractable by the current computer.

Removing motif overlaps may also give a potential speedup to the search. As argued, this is not trivial and needs further investigation.

We also included a method comparing subgraph instances to each other, for every motif. gSpan considers all subgraph instances equal, and score tells something about the similarity between the instances and may be usable for the user during result analysis. Further improvements to this implementation may be to include instance similarity tests during a search, pruning away motifs representing less similar instance sets.

5.5 About the development of NeMi

This project was developed jointly by two programmers. The whole project was developed in Java using both Eclipse and IntelliJ, we used Subversion for source control. Cytoscape version 2.2 was the target for our plugin. The testing code was written in Python and proved effective for parsing output files and plotting data.

We developed the basic features of the plugin very early in the process. Iterativley adding new features and improving on previous versions. The core of the algorithms was developed using pair programming. This enabled us to work much faster on the complex parts of the code. The code base now contains over 10000 lines of code, all commented using Javadoc.

When programming we often encountered the problem of state explosion. This led to lots of memory allocation. To counter this problem we have inserted code that enables the mining algorithms and motif finder to fail gracefully in case of memory allocation errors. Also we have used some tricks for saving memory and to cut recursions early (like the DFS Apriori). We have used a lot of recursive calls in our code. A way to reduce memory load is to convert the recursions to iterations. This is reserved for future work.

Another problem we often encountered was the badly documented Cytoscape API. Some of the API is documented, and some isn't. Some documentataion of the graph API is available on the GINY[1] webpage, but it is sparsely documented. Some of the functions in Cytoscape didn't operate as expected, for example we had to design our own functions for finding neighbors of nodes.

Most of the rest we had to look through the source code of Cytoscape itself to find solutions. The Gene Ontology annotation API is not documented at all, and we had to figure it out by trial and error. The GUI of the plugin was developed in SWING, and we spent a lot of time debugging the GUI code. The reason the CSA Neighborhood Matching protein similarity measure is implemented as a clusterer is a limitation in the GUI for configuring our similarity measures. More work could be done to make the GUI better and more understandable. Especially more work could be done to enable investigation of the clusters.

It was our hope that the plugin could be included on the Cytoscape plugin page. We posted our plugin on the `cytoscape-discuss` mailinglist. Chris Workman, who is a developer of Cytoscape, responded with some bug reports and asked for more info. The plugin isn't on the plugin page as of June 9th 2006, it is still in "beta" state. We hope that if biologists find this tool useful that it can be included on the Cytoscap plugin page in the future.

[1]GINY is the graph API incorporated into Cytoscape

Chapter 6

Summary and Conclusion

The work presented in this report has introduced ways of investigating link structure data in biological networks. We have looked at 3 problems and proposed solutions to them. Comparing proteins based on neighborhood information, finding frequent sub-patterns in a network and finding common patterns in several networks. We have developed similarity measures, clusterers and algorithms to achieve this goal and made it available in an freely available plugin to Cytoscape.

We have continued the work of Braute and Rødsjø (2005) and added a better ranking of the neighborhoods. The central part of the Protein Finder is finding the maximum bipartite matching between two nodes' neighborhoods. We have implemented the high-performing Cost Scaling Algorithm in Java to enable platform independence. A set of similarity measures have been implemented, both through the Gene Ontology protein annotations, and Transcription Factor data.

Our proposed algorithm for finding a given motif in a larger network, Motif finder, searches for near-isomorphic topology match using node similarity matrices for the scoring of a match. It searches depth first and includes a user specified branch & bound factor. The algorithm proved to be of great value throughout testing and analysis of the graph miners.

We implemented two distinct approaches mining sub-graph patterns in biological interaction networks. First, we implemented the Apriori algorithm, a general mining technique adopted to graphs. By preprocessing the data using our similarity measure implementations, we clustered the nodes in order to assign node labels. Near matching is accomplished by allowing nodes to occur in multiple clusters. We improved the Apriori algorithm allowing mining single graphs, and improved its speed by converting it to a depth-first hybrid. However, Apriori can only discover induced subgraphs and also it became too slow mining larger subgraphs.

gSpan mines subgraphs in a depth first manner, searching for connected subgraphs only, avoiding the massive candidate generation explosion. Benefiting our experience from the Apriori implementation, we enable single graph mining and near matching using clustering. We added several parameters to the algorithm involving restrictions in instance overlapping and maximum sub-graph size. To reduce search space and lower the motif's specificity, we removed backward edges. This made the algorithm faster and more adapted for biological searches.

We have compared the speed and verified our implementations correctness. By comparing the Motif finder and the graph miners, we show that our using clustering through similarity measures give the expected output in our experiments. All our implementations are freely available online as NeMi, a plug-in for Cytoscape, a well known tool for visualization and analysis of bio-molecular interaction data.

We have found that graph mining indeed can give interesting results for the problems presented to us. Comparing Apriori with gSpan we conclude that gSpan is both faster and gives better results than Apriori. It still remains to investigate the biological interestingness of our approaches. This was out of scope for our thesis. Such investigation may demand more sofisticated clustering techniques or similarity measures, which may easily be added to our plug-in.

Appendix A

Bibliography

Bibliography

Michael Ashburner, Catherine A. Ball, Judith A. Blake, David Botstein, Heather Butler, J. Michael Cherry, Allan P. Davis, Kara Dolinski, Selina S. Dwight, Janan T. Eppig, Midori A. Harris, David P. Hill, Laurie Issel-Tarver, Andrew Kasarskis, Suzanna Lewis, John C. Matese, Joel E. Richardson, Martin Ringwald, Gerald M. Rubin, and Gavin Sherlock. Gene ontology: tool for the unification of biology. *Nature Genetics*, 25: 25–29, 2000.

G. Bader, I. Donaldson, C. Wolting, B. Ouellette, T. Pawson, and C. Hogue. Bind - the biomolecular interaction network database, 2001. URL `citeseer.ist.psu.edu/bader01bind.html`.

Petter Braute and Jorg Rødsjø. Protein function prediction using annotated protein-protein interaction networks. Master's thesis, Norwegian University of Science and Technology, 2005. URL `neoplex.org/school/uni/master_braute_og_rodsjo.pdf`.

Diane J. Cook and Lawrence B. Holder. Substructure discovery using minimum description length and background knowledge. *Journal of Artificial Intelligence Research*, 1:231–255, 1994. URL `citeseer.ist.psu.edu/article/cook94substructure.html`.

Minghua Deng, Zhidong Tu, Fengzhu Sun, and Ting Chen. Mapping gene ontology to proteins based on protein–protein interaction data. *Bioinformatics*, 20(6):895–902, 2004. ISSN 1367-4803. doi: dx.doi.org/10.1093/bioinformatics/btg500.

Finn Drabløs. Personal communication. Several meetings in the period from 16th of January up to the 12th of June., 2006.

Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979. ISBN 0716710447.

Andrew V. Goldberg and Robert Kennedy. An efficient cost scaling algorithm for the assignment problem. *Math. Program.*, 71(2):153–177, 1995. ISSN 0025-5610. doi: http://dx.doi.org/10.1007/BF01585996.

Laurie J. Heyer, Semyon Kruglyak, and Shibu Yooseph. Exploring Expression Data: Identification and Analysis of Coexpressed Genes. *Genome Res.*, 9(11):1106–1115, 1999. doi: 10.1101/gr.9.11.1106. URL `http://www.genome.org/cgi/content/abstract/9/11/1106`.

Haiyan Hu, Xifeng Yan, Yu Huang, Jiawei Han, and Xianghong Jasmine Zhou. Mining coherent dense subgraphs across massive biological networks for functional discovery. *Bioinformatics*, 21(1):213–221, 2005. doi: 10.1093/bioinformatics/bti1049. URL `http://bioinformatics.oxfordjournals.org/cgi/content/abstract/21/suppl_1/i213`.

Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. Complete mining of frequent patterns from graphs: Mining graph data. *Mach. Learn.*, 50 (3):321–354, 2003. ISSN 0885-6125. doi: http://dx.doi.org/10.1023/A:1021726221443.

Harold W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistic Quarterly*, 2:83–97, 1955.

Steven Maere, Karel Heymans, and Martin Kuiper. Bingo: a cytoscape plugin to assess overrepresentation of gene ontology categories in biological networks. *Bioinformatics*, 21(16):3448–3449, 2005. doi: 10.1093/bioinformatics/bti551. URL `http://bioinformatics.oxfordjournals.org/cgi/content/abstract/21/16/3448`.

Paul Shannon, Andrew Markiel, Owen Ozier, Nitin S. Baliga, Jonathan T. Wang, Daniel Ramage, Nada Amin, Benno Schwikowski, and Trey Ideker. Cytoscape: A Software Environment for Integrated Models of Biomolecular Interaction Networks. *Genome Res.*, 13(11):2498–2504, 2003. doi: 10.1101/gr.1239303. URL `http://www.genome.org/cgi/content/abstract/13/11/2498`.

Roded Sharan. Identification of protein complexes by comparative analysis of yeast and bacterial protein interaction data. *Journal of Computational Biology*, 12(6):835–846, 2005. URL `http://www.liebertonline.com/doi/abs/10.1089/cmb.2005.12.835`.

Steven S. Skiena. *The algorithm design manual*. Springer-Verlag New York, Inc., New York, NY, USA, 1998. ISBN 0-387-94860-0.

Mehmet Koyuturk Yohan Kim Shankar Subramaniam Wojciech Szpankowski and Ananth Grama. An efficient algorithm for detecting frequent subgraphs in biological networks, (with m. koyuturk, and a. grama). *Bioinformatics*, pages 200–207, 2005. URL `http://www.cs.purdue.edu/homes/spa/papers/mining.pdf`.

Takashi Washio and Hiroshi Motoda. State of the art of graph-based data mining. *SIGKDD Explor. Newsl.*, 5(1):59–68, 2003. ISSN 1931-0145. doi: http://doi.acm.org/10.1145/959242.959249.

Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. In *ICDM '02: Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM'02)*, page 721, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1754-4.

Motoda H Yoshida K and Indurkhya N. Graphbased induction as a unified learning framework. *J. of Applied Intelligence*, 4(3):297–328, 1994.

Appendix B

Users guide to NeMi

NeMi

Neighborhood Mining

NeMi is short for "Neighborhood mining" and is plug-in for Cytoscape for investigating the link relations protein-protein interaction networks. NeMi has three major components; Protein Finder, Motif Finder and gSpan network mining. Here is a rough overview of these components. The most recent version of NeMi can be found on the web page `http://www.idi.ntnu.no/~sundsdal/nemi/`.

B.1 Installation

To install the plug-in download nemi.zip from `http://www.idi.ntnu.no/~sundsdal/nemi/nemi.zip` and unzip to the "plugin" sub-folder in your Cytoscape program folder[1]. The plug-in will appear in the plug-ins list in Cytoscape upon restart. The zip also contains `bindings.txt`,

[1]Usually C:\Program Files\Cytoscape-v2.2

the ChIP-chip data from (Heyer et al., 1999), used by the TF-clusterer and the TF-comparer. This file must be placed in the Cytoscape program folder.

B.2 History

June 12th - Release
- Added stable and randomized iterative clusterer
- gSpan scores results
- All tools show status info.
- Memory heap space errors are now less frequent.

May 6th - Beta 3
- Added CSA neighborhood clusterer
- Added memory heap space full dialog
- Added help menu items
- Removed Apriori graph miner. Outperformed by gSpan on all tests.

May 27th - Beta 2
- Added transcription factor comparer and clusterer.
- Statistics now show GO info.

May 25th - Beta 1
- Added Apriori graph miner
- Made statistics for miners available after each run
- GUI now has status info

May 18th - Alpha 3
- Protein Finder scores with p-value from resampling test

May 12th - Alpha 2
- Updated release to include Motif Finder.

May 7th - Alpha 1
- Initial release

B.3 Protein similarity

All tools in this plugin utilize protein similarity measures. A protein is usually annotated by the Gene Ontology categories. A GO category can have parent categories and subcategories. A protein annotated with a GO category will therefore have a set of parent categories. When comparing two proteins their common parent categories are found. The common parents indicate how similar the two proteins are. If a they have common parents high up in the GO hierarchy this means that they have lower similarity than if they have common parents far down in the hierarchy. Also if many proteins have the same parents they may not be as similar if only two proteins share the same parent category. Therefore there are two ways of utilizing the GO annotation for protein similarity. Either using the deepest common parent or the least likely common parent.

We have also included a Transcription Factor comparer with data from (Heyer et al., 1999). For flexibility the user can mix and match as much as he/she pleases using a weighted sum of the similarity scores.

The "Protein comparer" pane lets you select a node in a network and see the calculated similarity.

B.4 Motif Finder

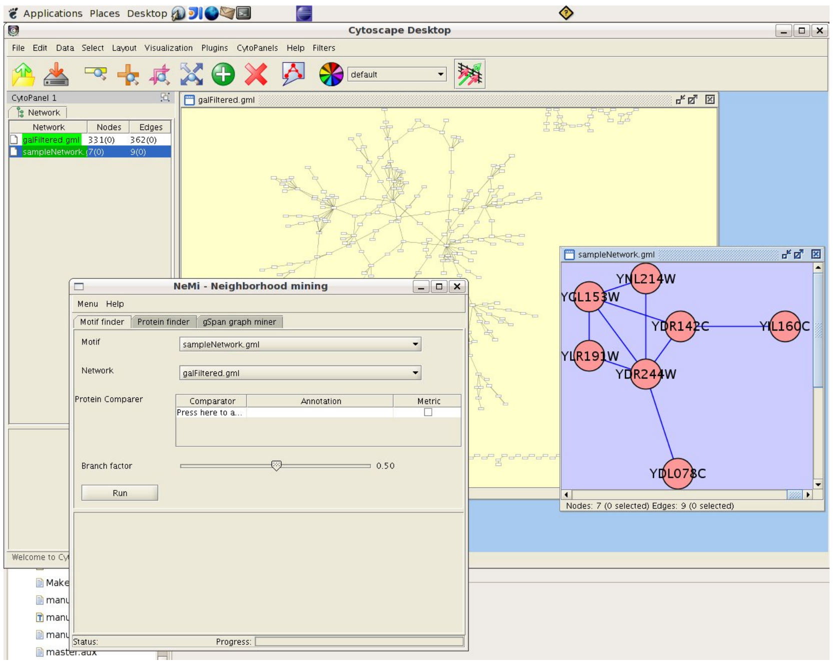
Motif Finder lets you search for one network (motif) in another network. Start Cytoscape and open two networks, one representing the motif and another where you want to search for this motif. It is based on a depth first search for a matching spanning tree. In this module no clustering is done. The score of a match is the sum of the similarity measures for the protein matching.

B.4.1 How to use

You need to open two networks in Cytoscape before starting the plug-in. One motif network and one network where you wish to search for that network, see figure B.1

Select the kind of protein similarity measure you wish to use. Then select annotation information. You can add as many similarity measures as you like and assign weights to each measure.

Then click "run". The results will appear at the bottom of the screen as seen in figure B.2. Clicking on the line will highlight (select) the matched nodes in the network. If the comparison runs slowly the the branch-and-bound should be tighter. Adjusting the branch factor up will yield less results, but should be faster. Results can be saved to file by clicking "Save results".



Network	Nodes	Edges
galFiltered.gml	331(0)	362(0)
sampleNetwork	7(0)	9(0)

Motif: sampleNetwork.gml

Network: galFiltered.gml

Branch factor 0.50

Nodes: 7 (0 selected) Edges: 9 (0 selected)

Figure B.1: ExampleNetwork and galFiltered in motif finder

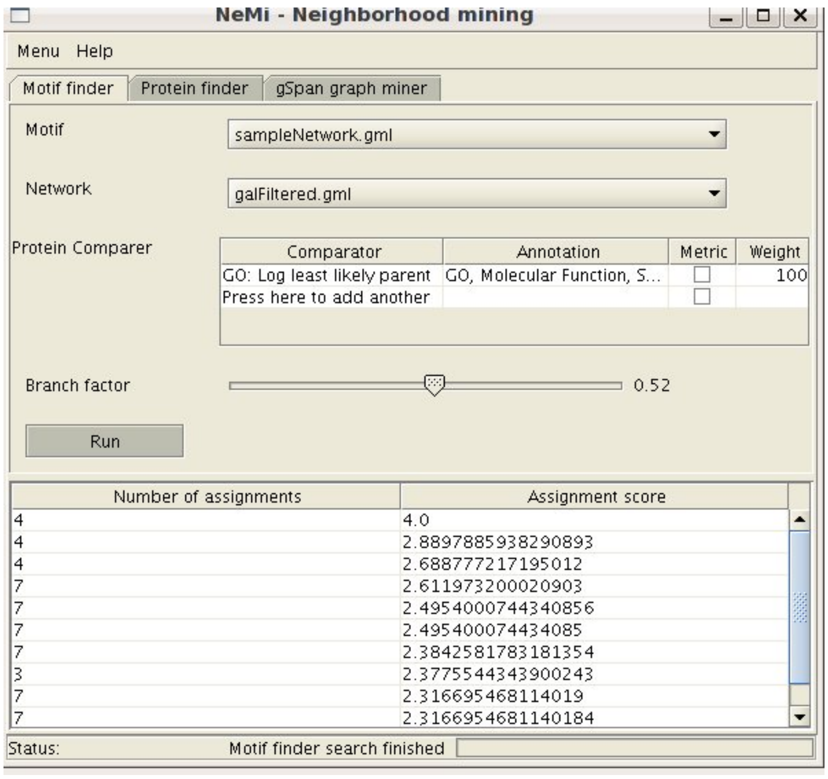
B.5 Protein Finder

The Protein Finder is based on the principle that the function of a protein may be induced by the proteins it interacts(Deng et al., 2004). If two proteins have similar neighborhoods, then there is a high probability that the two share similar functionality.

The Protein Finder lets you select a protein and find other proteins with similar "neighborhoods". It returns proteins with scores listed in descending order. The score is based on an optimal matching between the neighbor set of the two proteins. A neighbor set of a protein is all proteins interacting with that protein. The proteins in the two neighbor sets are compared all to all and a similarity score is obtained for each pair. An optimal matching has a maximal sum of scores. The score may be replaced with a p-value for statistical relevance using re-sampling statistics.

B.5.1 How to use Protein Finder

Before starting the plug-in you must open a network in Cytoscape. After starting the plug-in switch to the "Protein Finder" tab. Now you can select a protein in the network to do neighborhood matching. Create your



Motif: sampleNetwork.gml

Network: galFiltered.gml

Protein Comparer

Comparator	Annotation	Metric	Weight
GO: Log least likely parent	GO, Molecular Function, S...	☐	100
Press here to add another		☐	

Branch factor 0.52

Number of assignments	Assignment score
4	4.0
4	2.8897885938290893
4	2.688777217195012
7	2.611973200020903
7	2.4954000744340856
7	2.495400074434085
7	2.3842581783181354
3	2.3775544343900243
7	2.316695468114019
7	2.3166954681140184

Status: Motif finder search finished

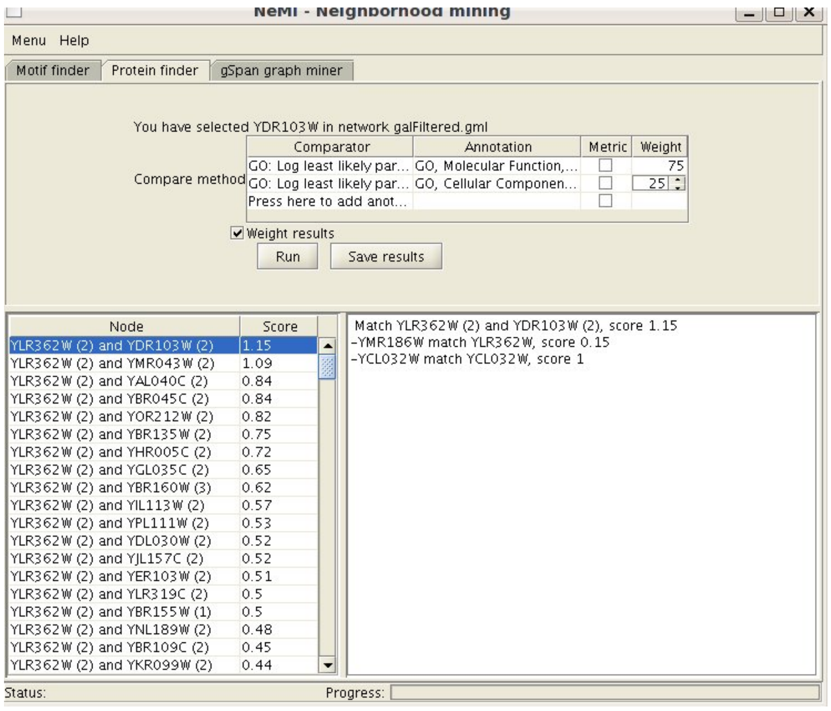
Figure B.2: The matching list

own compare method by using the compare method table. If you wish to weight the score by re-sampling the neighborhoods select "weight results" toggle-box.

Now click run for the matching to proceed. This can take some time, the progress bar shows the current activity. The results will show up in the left bottom panel as shown in figure B.3. Selecting a result will show a textual description of the match to the right, and the protein will also be highlighted(selected) in the network. If you wish to save the matching data to file click "Save results"

B.6 gSpan graph miner

gSpan is a graph mining algorithm by Yan and Han (2002) that we have implemented for protein interaction graphs. It finds frequent sub-graphs in one or several networks. It is designed to look for protein interaction patterns that are common to a set of networks (different species) or a pattern that reoccurs frequently in one network.



You have selected YDR103W in network galFiltered.gml

Compare method

Comparator	Annotation	Metric	Weight
GO: Log least likely par...	GO, Molecular Function,...	☐	75
GO: Log least likely par...	GO, Cellular Componen...	☐	25
Press here to add anot...		☐	

☑ Weight results

Node	Score
YLR362W (2) and YDR103W (2)	1.15
YLR362W (2) and YMR043W (2)	1.09
YLR362W (2) and YAL040C (2)	0.84
YLR362W (2) and YBR045C (2)	0.84
YLR362W (2) and YOR212W (2)	0.82
YLR362W (2) and YBR135W (2)	0.75
YLR362W (2) and YHR005C (2)	0.72
YLR362W (2) and YGL035C (2)	0.65
YLR362W (2) and YBR160W (3)	0.62
YLR362W (2) and YIL113W (2)	0.57
YLR362W (2) and YPL111W (2)	0.53
YLR362W (2) and YDL030W (2)	0.52
YLR362W (2) and YJL157C (2)	0.52
YLR362W (2) and YER103W (2)	0.51
YLR362W (2) and YLR319C (2)	0.5
YLR362W (2) and YBR155W (1)	0.5
YLR362W (2) and YNL189W (2)	0.48
YLR362W (2) and YBR109C (2)	0.45
YLR362W (2) and YKR099W (2)	0.44

```
Match YLR362W (2) and YDR103W (2), score 1.15
-YMR186W match YLR362W, score 0.15
-YCL032W match YCL032W, score 1
```

Figure B.3: protein Finder with results

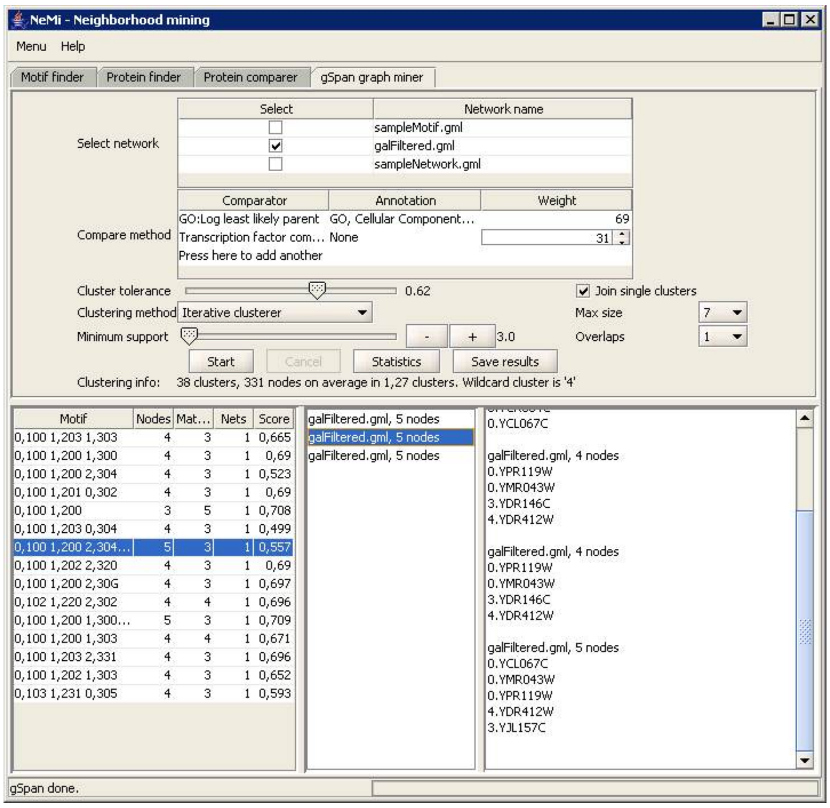
A protein interaction network consists of unique proteins. To find patterns using data mining some proteins have to be considered equal. This is done through clustering. Proteins with similarity over a certain threshold are assigned a cluster. The threshold is called the tolerance of the cluster. Note that high clustering tolerance will result in less clusters, and there is a higher probablility that a node exists in more clusters which makes the search space larger. Searching large networks, the clustering tolerance may be set to a low value to ensure high performance. The support level defines how many matches a pattern [2] must have for inclusion in the result set.

After the clustering a new interaction network is created behind the scenes. The identifier for the nodes in the new network is cluster label + protein identifier. A protein contained in several clusters will result in several nodes in the new network, one for each cluster. Nodes starting with the same cluster label are treated as equal. The mining will result in a code or motif which has the same structure as a spanning tree. The matches of this motif can be viewed by clicking on the list of results.

[2]A pattern is the same as a motif in the plug-in.

B.6.1 How to use gSpan graph mining

Open a network or a set of networks in Cytoscape before starting the plug-in. Switch to "gSpan graph miner" and select the networks. If you select more than one network the graph mining will ensure that a pattern at least exists in all networks. Create your similarity measure by using the compare method table. Cluster tolerance decides how similar a set of nodes must be to be included in the same cluster. Minimum support is set by the slider.If you wish to restrict the size of the network motif to increase motif it can be done with the "Max size" combobox. Adjusting the allowed number of nodes overlapping can be done with the "Overlaps" combobox. See figure B.4.


NeMi - Neighborhood mining
Menu Help
Motif finder | Protein finder | Protein comparer | gSpan graph miner
Start
Cancel
Statistics
Save results
-
+

Select network

Select	Network name
☐	sampleMotif.gml
☑	galFiltered.gml
☐	sampleNetwork.gml

Compare method

Comparator	Annotation	Weight
GO:Log least likely parent	GO, Cellular Component...	69
Transcription factor com...	None	31
Press here to add another		

Cluster tolerance 0.62 ☑ Join single clusters
Clustering method Iterative clusterer Max size 7
Minimum support 3.0 Overlaps 1

Clustering info: 38 clusters, 331 nodes on average in 1,27 clusters. Wildcard cluster is '4'

Motif	Nodes	Mat...	Nets	Score
0,100 1,203 1,303	4	3	1	0,665
0,100 1,200 1,300	4	3	1	0,69
0,100 1,200 2,304	4	3	1	0,523
0,100 1,201 0,302	4	3	1	0,69
0,100 1,200	3	5	1	0,708
0,100 1,203 0,304	4	3	1	0,499
0,100 1,200 2,304...	5	3	1	0,557
0,100 1,202 2,320	4	3	1	0,69
0,100 1,200 2,30G	4	3	1	0,697
0,102 1,220 2,302	4	4	1	0,696
0,100 1,200 1,300...	5	3	1	0,709
0,100 1,200 1,303	4	4	1	0,671
0,100 1,203 2,331	4	3	1	0,696
0,100 1,202 1,303	4	3	1	0,652
0,103 1,231 0,305	4	3	1	0,593

galFiltered.gml, 5 nodes
galFiltered.gml, 5 nodes
galFiltered.gml, 5 nodes

```
0.YCL067C

galFiltered.gml, 4 nodes
0.YPR119W
0.YMR043W
3.YDR146C
4.YDR412W

galFiltered.gml, 4 nodes
0.YPR119W
0.YMR043W
3.YDR146C
4.YDR412W

galFiltered.gml, 5 nodes
0.YCL067C
0.YMR043W
0.YPR119W
4.YDR412W
3.YJL157C
```

gSpan done.

Figure B.4: Gspan graph mining pane

Press "Start" to commence graph mining. When graph mining is done a new network will appear in Cytoscape with prefix "NeMitemp" and postfix "(nemi)".Mining results will appear in bottom half of the window. Selecting a match line will highlight the match in the two networks. This can be seen by bringing the main Cytoscape window to front. All sub-graph instances will appear in the middle, and clicking an instance will highlight it in its corresponding network. Detailed instance information is given in the rightmost panel.

B.7 Apriori graph miner

Apriori is a graph mining algorithm by Inokuchi et al. (2003) that we have implemented for protein interaction graphs. It finds frequent sub-graphs in one network.

Apriori does almost the same as gSpan only it finds connected induced sub-graphs, while gSpan finds connected sub-graphs. In most applications gSpan will be faster and better. Apriori is included mainly for comparison. It can be found in the "Apriori graph miner" panel.

B.8 Mining statistics

After a mining run is finished, a statistic window for the clustering can be found by the "Statistics" button. See figure B.5.

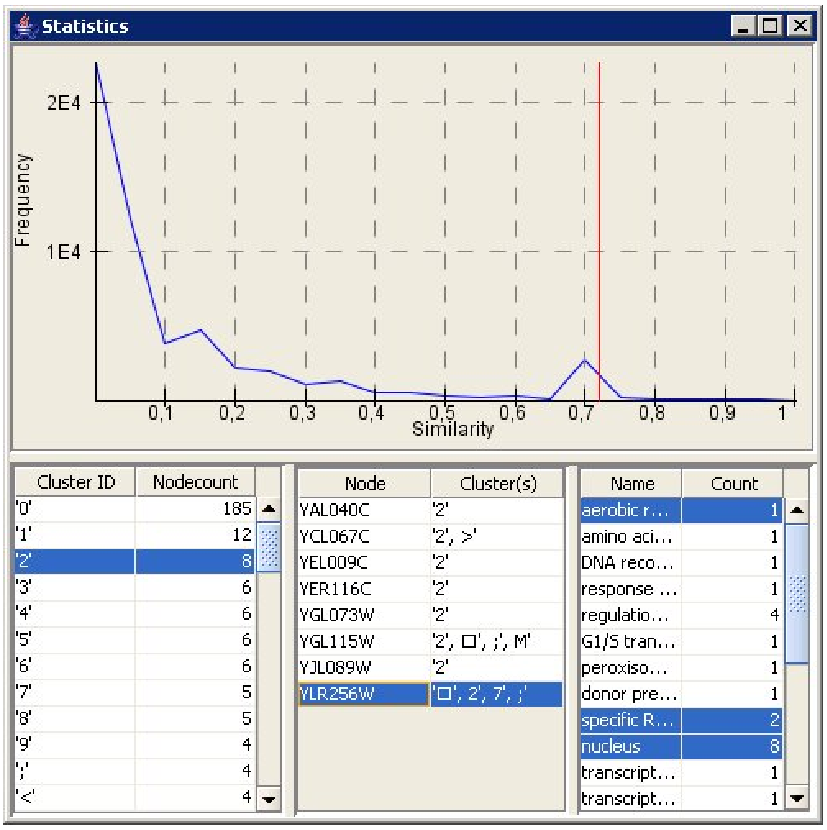
The graph is a frequency plot the comparer. It shows the similarity distribution. Lower left is a list of clusters. Clicking them gives a list of nodes in the cluster. In the lower right you can the Gene Ontology categories associated with the nodes in the cluster.

B.9 Some final words

This plug-in is a product of a Master thesis by Kristoffer Stenersen and Sverre Sundsdal. We are computers scientist, not biologists and we need input. Please contact us if you find this plug-in interesting or if you want more information. Feature requests, bug reports and usage reports are greatly appreciated. NeMi, Neighborhood Mining in biological networks is also a name of a norwegian comic strip, Nemi.

- Kristoffer Stenersen - kristoffer.stenersen@gmail.com
- Sverre Sundsdal - sundsdal@gmail.com

June 16th 2006



Cluster ID	Nodecount
'0'	185
'1'	12
'2'	8
'3'	6
'4'	6
'5'	6
'6'	6
'7'	5
'8'	5
'9'	4
';'	4
'<'	4

Node	Cluster(s)
YAL040C	'2'
YCL067C	'2', >'
YEL009C	'2'
YER116C	'2'
YGL073W	'2'
YGL115W	'2', □', ;', M'
YJL089W	'2'
YLR256W	'□', 2', 7', ;'

Name	Count
aerobic r...	1
amino aci...	1
DNA reco...	1
response ...	1
regulatio...	4
G1/S tran...	1
peroxiso...	1
donor pre...	1
specific R...	2
nucleus	8
transcript...	1
transcript...	1

Figure B.5: Clustering statistics after mining

Appendix C

How-To

In this appendix we will describe how to add a similarity measure and a new clustering algorithm to NeMi. In order to add code to the project you need Java 5.0 SDK, a text editor and the libraries and source code to NeMi. This should be available together with this thesis. If you make changes to NeMi source code it needs to be recompiled and packed into a .jar file. To compile a nemi.jar file we recommend you run the `ant jar` command. The file `basic.properties` must be available. Example files for Linux and Windows are included in the source distribution. To copy the jar to Cytoscape run `ant deploy`.

C.1 Add a similarity measure

Protein comparers should be put in the package `nemi.compare`. It should extend the abstract class `ProteinCompare` and then be inserted into the `ProteinComparerTable` in `nemi.compare.gui`.

First we show the interface of the comparer:

```
public abstract class ProteinCompare {
    /**
     * Returns a value for the similarity between two
     * proteins. A similarity score of 1.0 means they are
     * equal (or max similar), and 0.0 means total
     * dissimilarity.
     *
     * @return similarity between 0 and 1.
     **/
    abstract public double similarity(NetNode nn1,
                NetNode nn2);
    /**
     * This method is here so that implementers should
     * remember to add a static field "name" for the GUI.
     **/
    abstract public String getName();
}
```

The `NetNode` objects are a combination of Cytoscape network and a node:

```
public class NetNode {
    public CyNetwork net;
    public Node node;

    public NetNode(CyNetwork net, Node node) {
        this.net = net;
        this.node = node;
    }
[snip]
}
```

It is important that all ProteinComparer has a static String field `name` that contains its name. This is used in the GUI. As an example we here include a fictional similarity measure based on the String edit-distance of the `identifier`:

```
public class EditdistanceComparer extends ProteinCompare {
  public static String name=''Edit distance comparer'';
  public double similarity(NetNode nn1,
      NetNode nn2){
    int maxlength = nn1.node.getIdentifier().length();
    maxlength = Math.max(maxlength,
        nn2.node.getIdentifier().length());
    int editdistance = computeEditditstance(
        nn1.node.getIdentifier(),
        nn2.node.getIdentifier());
    return (double)editdistance/(double)maxlength;

  }
  public String getName(){
    return name;
  }
}
```

The method `int computeEditDistance(String, String)` (not shown) gives a number for the number of edits necessary to convert one string to another . This will always be 0 or above. Then we normalize each score by `maxlength` to make certain that the result is less than 1.0. Saving/Caching the result between runs to increase speed later on is not necessary because it is taken care of outside the comparer (in `CompositeComparer`). Other examples of comparers can be found in the package `nemi.compare`.

To include a comparer in the GUI it needs to be included in the `ProteinComparerTable`. First it needs to be included in the static list of classes, `comparers`. We add it to the end of the list like this:

```
public static final Class[] comparers =
  new Class[]{
    nemi.compare.TransFacCompare.class ,
    nemi.compare.DeepestCommonGOCategory.class ,
    nemi.compare.LeastLikelyGOParent.class ,
    nemi.compare.LogLeastLikelyGOParent.class ,
    nemi.compare.EditDistanceComparer.class
  };
```

Then we need to include code to add initiate this comparer in `getComparer`. Add this code to the end of the ifs and else ifs in the for loop:

```
[snip]
    } else if (model.comparer.get(i).equals(
          EditDistanceComparer.class)) {
      EditDistanceComparer c = new EditDistanceComparer();
      comps.add(c);
    }
[snip]
```

Now your comparer should be ready to go. If your comparer requires Gene Ontology data this is available in the table model. We refer to the source code on how to accomplish this.

C.2 Add a clusterer

The clustering code is found in package `nemi.cluster`. First you should familiarize yourself with the `Cluster` class. Here we show the central methods:

```
public class Cluster implements Comparable<Cluster> {
  /* a counter to ensure that by default all
     clusters have a different id */
  private static char counter = '0';
  public char id = counter++;
  List<NetNode> repset ;

  /**
  * Initialize a cluster with a representative list
  */
  public Cluster(List<NetNode> nodes) {
    repset = new Vector<NetNode>(nodes);
  }
  /**
  * get a copy of this cluster
  */
  public Cluster clone() {
    return new Cluster(repset);
  }
  /**
  * Get the representatives for this cluster
  */
  public Collection<NetNode> getRepNetNodes() {
    return this.repset;
  }
  /**
  * Count the number of nodes in this cluster.
  */
  public int size() {
    return repset.size();
  }
  public void merge(Cluster w) {
    repset.addAll(w.repset);
  }
  /**
  * Compare cluster by id only.
  */
  public int compareTo(Cluster o) {
    return this.id - o.id;
  }
  /**
  * add a node to the cluster.
  */
  public void add(NetNode netNode) {
    repset.add(netNode);
  }
}
```

Notice that all a cluster really is is a collection of Nodes and the id of a cluster. Comparisons between clusters here are only based on id. Many clustering algorithm represents clusters with a “center”. If this is to be done the `Cluster` class must be sub classed. Some methods for measuring quality, similarity to nodes and other clusters are available in the `QualityAssesser` class.

As an example we include a fictional clusterer that clusters nodes in groups with the size based on the Fibonacci sequence:

```java
public class FibonacciClusterer implements Clusterer {
  public List<Cluster> cluster(Collection<CyNetwork>
        networks, boolean joinSingleClusters){
    List<Cluster> clusters = new Vector<Clusters>();
    int f1 = 0;
    int f2 = 1;
    int i = 0;
    Cluster c;
    for (CyNetwork net : networks) {
      for (Iterator<Node> nodeIter = net.nodesIterator();
          nodeIter.hasNext();) {
        if(++i == f2){
          int temp = f2;
          f2 = f1+f2;
          f1 = temp;
          c = new Cluster(new Vector<Node>());
          clusters.add(c);
        }
        Node n = nodeIter.next()
        c.add(new NetNode(net, n));
      }
    }
    char maxChar = 0;
    i = 0;
    for(Cluster clust : clusters){
      if (i < readableCharacters.length) {
        clust.id = readableCharacters[i++];
        if (clust.id > maxChar) maxChar = clust.id;
      } else {
        clust.id = ++maxChar;
      }
    if(joinSingleClusters){
      Cluster single = new Cluster(new Vector<Node>());
      single.id='?';
35       for(Iterator<Cluster> cIter = clusters.iterator();
              cIter.hasNext()){
37         Cluster clust = cIter.next();
           if(clust.size()!=1) continue;
39         single.merge(clust);
           cIter.remove();
41       }
         clusters.add(single);
43     }
       return clusters;
45   }
   }
```

This example clusterer goes through the basic steps of clustering. In line 8-21 clusters are created according to some measure. In line 22-31 the clusters are relabeled to ensure that they use the `readableCharacters` defined in the parent class `Cluster`. If not, some extra characters are added by incrementing `maxChar`. In lines 32-42 single clusters are joined to a wild-card cluster with label "?".

Then the code to activate this clusterer must be included in the `ClustererList` class. Insert a static variable to include the name of the clusterer:

```
 public static String fibCluster = "Fibonacci␣clusterer";
```

and also add it as an element in the list:

```
    public ClustererList() {
       this.addElement(itClust);
       this.addElement(rItClust);
       this.addElement(tfCluster);
       this.addElement(csaCluster);
       this.addElement(fibCluster);
    }
```

And then it needs to be added in the `cluster` method in the if and else if statements:

```
[snip]
   } else if((getSelectedItem().equals(
           ClustererList.fibCluster)) {
       FibonacciClusterer clust = new FibonacciClusterer();
       clusters = clust.cluster(networks, joinSingle);
   } else {
[snip]
```

Now you have created a clusterer. We encourage you to look closer at the other clusterers for ideas.