HÖGSKOLAN SKÖVDE

Institutionen för kommunikation och information
Examensarbete i datalogi 20p
C-nivå
Vårterminen 2005

Biologically plausible visual representation of modular decomposition

Jonas Rahm

HS-IKI-EA-05-107

Biologically plausible visual representation of modular decomposition

Examensrapport inlämnad av Jonas Rahm till Högskolan i Skövde, för Kandidatexamen (B.Sc.) vid Institutionen för kommunikation och information.

2005-06-06

Härmed intygas att allt material i denna rapport, vilket inte är mitt eget, har blivit tydligt identifierat och att inget material är inkluderat som tidigare använts för erhållande av annan examen.

Signerat: _____

Handledare för examensarbetet: Zelmina Lubovac

Biologically plausible visual representation of modular decomposition

Jonas Rahm

Abstract

Modular decompositions of protein interaction networks can be used to identify modules of cooperating proteins. The biological plausibility off these modules might be questioned though. This report describes how a modular decomposition can be completed with semantic information in the visual representation. Possible methods for creating modules of functionally related proteins are also proposed in this work. The results show that such modules, with advantage can be combined with modules from a graph decomposition, to find proteins that are likely to cooperate to perform certain functions in organisms.

Keywords: Modular decomposition, Biological knowledge, Yeast two-hybrid technology, The Gene Ontology.

Contents

1 Introduction

In this work, modular decomposition is applied to identify the set of functional modules, which are then related to domain knowledge in terms of Gene Ontology.

Modules in a protein network can be described as sub-systems of the network. There are several different definitions of functional modules. Hartwell et al. (1999) define a module as "a discrete entity whose function is separable from those of other modules". A module can further be defined as a group of elements that cooperate to achieve a common goal. Modular decomposition of protein interaction networks is important because it might be used to find functional modules, where the proteins in a module are likely to cooperate to perform a certain biological function. Hence, a modular decomposition of a protein interaction network can be used to increase understanding of biological processes by, for example, studying how knocking out groups of proteins instead of individuals may affect the phenotype.

Gagneur et al. (2004) proposed a method where a graph representation of a protein interaction network, with proteins as vertices and interactions as edges between vertices, can be decomposed into rooted trees where the internal nodes represents modules. The biological plausibility of the modules might be questioned though, because there are no objective criteria for how modules are identified in a protein interaction network. Furthermore, errors may be present in databases that store protein interaction data (Deng et al., 2003) and there are several protein interaction analysis techniques with different features and applications (Piehler, 2005). For example, some analysis techniques are applied *in vivo* and others *in vitro*. The semantics of the interactions from different analysis techniques varies too (Gagneur et al., 2004). Protein interaction data can, in some cases, be more reliable if interaction data sets from different techniques are compared (Zhong et al., 2003). But, even if the interactions in a protein interaction dataset are correct, it is not sure that the modular decomposition of a network gives a correct view of which proteins are cooperating to perform biological functions.

The purpose of this work is to develop a method that integrates domain knowledge with modular decomposition of protein interaction networks, in order to make it easier to assess the biological plausibility of the modules. In the integration procedure, minimum subsumer terms (according to the sub-ontology that describes molecular function in Gene Ontology) will be assigned to the modules in a modular decomposition. By giving a semantic perspective to the modular decomposition, the plausibility and the importance of the modules should be easier to study. A system that performs this integration and visualizes the resulting module tree will be implemented in this work. Some important protein clusters are then used as input to this system, in order to assess the plausibility of the decompositions from these networks. If functionally related proteins reside in the same modules, the modules should be biologically plausible. In order to verify the biological plausibility of modular decomposition in general, statistical methods might be used.

The report consists of the following chapters: *background*, *problem definition*, *method*, *algorithm development*, *system model*, *design and implementation*, *results* and *conclusion*. In the background chapter important concepts are introduced. In the problem definition chapter, the arguments that justify the relevance of the aim of the project are discussed in detail. The method chapter gives an overview of the method. In the algorithm development chapter, algorithms for the integration of semantic information with modular decomposition are described. The system model chapter

gives a model for the system that will be built. In the design and implementation chapter, a detailed model for how to develop the system will be given. In the results chapter, the results are described and analyzed, and in the final chapter the results and conclusions are summarized.

2 Background

In this chapter some important concepts are introduced and defined. Some related work in this area is also discussed.

2.1 Biological networks

In molecular biology, networks of biological elements are of great importance. An example of a biological network is a *protein interaction network*. Large-scale techniques in molecular biology, such as yeast two-hybrid system, have resulted in large data sets with pair-wise interactions between proteins. This information can be used to create protein interaction networks, represented by graphs. From a protein interaction network it is sometimes possible to find *clusters* of cooperating proteins. A cluster in a protein interaction network is a subset of the network. Highly interconnected nodes are often referred to as *hubs*. (Barabási and Oltvai, 2004)

Modules occur in many real-world networks (Barabási and Oltvai, 2004). In molecular biology the concept of a functional module is defined by Hartwell et al. (1999) as "a discrete entity whose function is separable from those of other modules". This means a module in a protein network is a group of proteins that cooperates with a distinct function as a common goal.

2.2 Modular decomposition

A modular decomposition is preferable to a protein interaction graph because it summarizes the structure of the graph. In graph-theory a module is "a set of nodes that have the same neighbors outside the module" (Gagneur et al., 2004, p. 2). A node *v* is a neighbor of a node *u* if there is an edge between node *u* and node *v*. The elements of a module can be substituted with a single representative node, called a *quotient*, because all nodes in a module share the same neighbors outside the module (Figure 1). Gagneur et al. (2004) states that the entire graph can be merged into a representative node if the quotients are iterated.

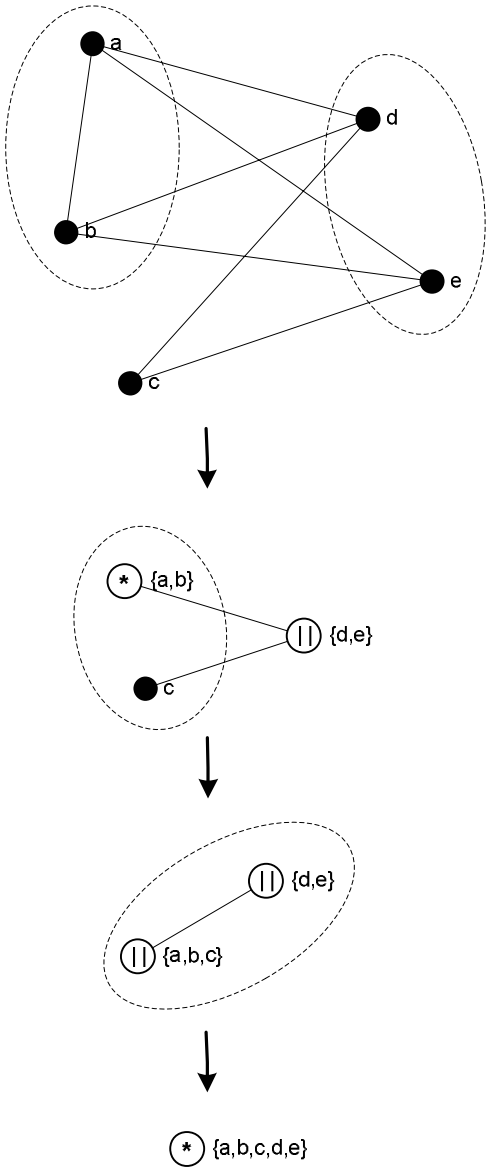

Figure 1: A graph and its modules. The modules in the graph are successively replaced with their according quotients. The parallel quotients are marked with || and the series quotients with *.

The modular decomposition of a graph is a tree of modules and proteins, with the modules as internal vertices and the proteins as leaves, as shown in shown in Figure 2. (Gagneur et al., 2004)

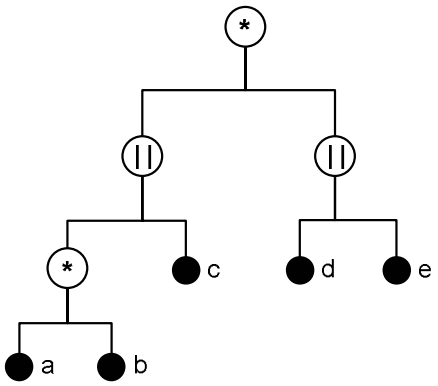

Figure 2: The modular decomposition of the graph in Figure 1. The root node represents the whole graph. The internal nodes represent the modules of the graph and the leaves the nodes in the graph.

2.2.1 Biological abstractions of modules

In protein interaction networks derived from yeast two-hybrid experiments, an interaction between two proteins means that the proteins can bind physically directly to each other (Gagneur et al., 2004). In a series module (Figure 3a) all of the proteins can bind directly to each other, while proteins in a parallel module (Figure 3b) cannot bind to each other. Prime modules (Figure 3c) are neither fully connected nor fully disconnected.

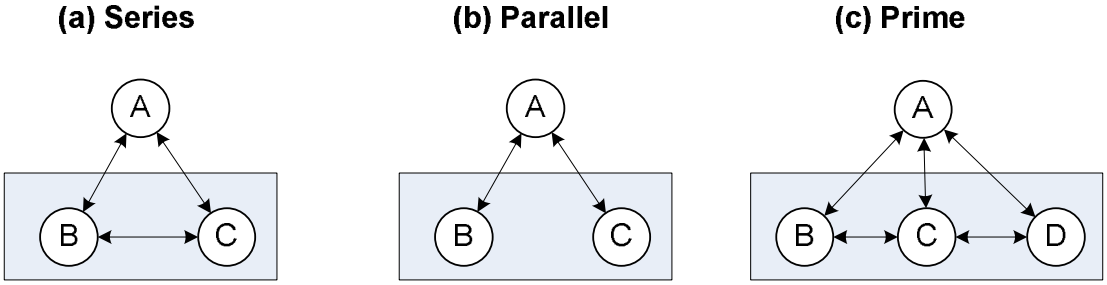

Figure 3: Interpretation of module labels for yeast two-hybrid experiments. In the series module (a), all proteins can bind directly to each other, while the proteins in a parallel module (b) are alternative binding partners to their common neighbors. In the prime module some proteins, but not all, can bind to each other.

2.3 The Gene Ontology

In addition to protein sequences and protein networks, there is lots of knowledge about protein function encoded in annotations, which may be written in natural scientific language. For computer processing these annotations are not always very suitable. A solution to this problem is the use of *ontological annotations*, such as *Gene Ontology* (The Gene Ontology Consortium, 2001). Gene Ontology is structured vocabulary that contains three sub-ontologies describing molecular function, biological process and cellular component (Lord et al., 2003).

GO holds about 11000 terms and concepts that are organized in a directed acyclic graph with *is-a* and *part-of* relationships. GO also supports evidence codes that can be used to classify the reliability of annotations. With GO-terms, annotations can be stored in databases in a structured way (Lord et al., 2003).

2.4 Existing tools

There are several tools that can be used to build the system. To identify modules *Modulor*[1] can be used and *GraphViz*[2], *JGo*[3], *Tom Sawyer Visualization*[4] and *Pajek*[5] are examples of tools and libraries for visualization. Modulor is a Java-tool (developed by Julien Gagneur), which generates the modular decomposition of a graph. The source code of Modulor follows the *MIT License*[6] and may therefore be

[1] http://www.mas.ecp.fr/labo/equipe/gagneur/module/module.html

[2] http://www.graphviz.org/

[3] http://www.nwoods.com/go/jgo.htm

[4] http://www.tomsawyer.com/

[5] http://vlado.fmf.uni-lj.si/pub/networks/pajek/

[6] http://www.opensource.org/licenses/mit-license.php

reused in this project. GraphViz and Pajek are the only visualization tools that are free of charge, from the above mentioned visualization tools. GraphViz has all the basic features that are needed to visualize module trees and with the well-documented DOT language used by GraphViz, it is easy to export a modular decomposition into a graph representation. The functionalities of GraphViz can also be integrated with Java-applications by using the Grappa-library. Pajek has a more advanced user interface than GraphViz and it has much built-in functionality.

3 Problem definition

In this chapter the motivation and aim of the project is described in detail. The aim is then divided into objectives.

3.1 Motivation

The biological plausibility of the modules in a modular decomposition of a protein interaction network might be questioned, because the underlying protein interaction data set might be incorrect (Deng et al., 2003; Piehler, 2005) and there are no objective criteria for how functional modules should be identified in a protein interaction network. If semantic information could be mapped to a modular decomposition, it would become easier to verify the relevance of modules. It might also be possible to use considerations about functional similarities as a part of the decomposition process. In this way new relationships among the proteins might be found.

3.2 Aims and objectives

The aim of this project is to develop algorithms that can be used to integrate information about biological function of proteins with modular decomposition of protein interaction networks. A system that implements these algorithms will also be developed, and from this system it will be possible to visualize the tree-like representation of modular decomposition together with functional information for the modules. To fulfill the aim, several objectives with sub-objectives, has been identified:

1. *Understanding fundamental concepts and algorithms.*
   1.1 Studying the literature that describes functional modules.
   1.2 Studying semantic similarity measures.
   1.3 Studying algorithms for modular decomposition.
2. *Development of algorithms and implementation of system.*
   2.1 Identifying appropriate protein interaction data sets.
   2.2 Choosing ontological annotation source.
   2.3 Developing the algorithms.
   2.4 Designing and implementing the system.
   2.5 Testing the system.
3. *Visualization and analysis*.
   3.1 Extending the system with visualization.
   3.2 Visualizing a subset of a network with its modules.
   3.3 Analyzing plausibility of the modules.

The main objectives and their sub-objectives are described in sections 2.2.1 to 2.2.3.

Objective 1: Understanding fundamental concepts and algorithms

It is important to gain a good understanding of the main topics in this work, i.e. modules and ontologies. Understanding of semantic similarity measures and existing

algorithms for modular decomposition is also very important. Besides modular decomposition by Gagneur, which uses the ordered vertex-partitioning algorithm developed by McConnell and Spinrad (2000), there is another algorithm that uses order extension described by Habib et al. (2004).

Objective 2: Development of algorithms and implementation of system

Before starting to develop the algorithm that integrates modular decomposition with information about the minimum subsumers of the modules (see Section 5.1.1), appropriate protein interaction datasets must be identified and annotation sources must be chosen. Functional similarity measures can either be integrated in the modular decomposition procedure, or they can be applied to the resulting modular decomposition from existing algorithm.

The system should be designed to be easy to understand, implement and extend. The system must generate output that can be used for both visualization and computer based processing. After the system has been implemented it has to be tested to avoid misleading conclusions in the analysis.

Objective 3: Visualization and analysis

The system that implements the algorithm for modular decomposition needs to be extended to visualize modules and their relationships. This can be obtained by using existing visualization tools or libraries. In order to generate results for analysis, subsets of a protein network and their modules, need to be visualized. In the visualization the modular decomposition should be visualized together with semantic information for the modules.

4 Method

In this chapter the method will be introduced, the minimum subsumer concept will be described and the data sets to be used will be presented.

4.1 Overview of the method

The aim of this project is to integrate knowledge about functional similarity for proteins with decompositions of protein interaction networks. Moreover the result needs to be visualized in an appropriate manner. The method that is used to meet the aim will be described in detail in chapters 4-6. The process of software development has to be iterative, because there is no exact specification for how the decomposition will be made or for how the results will be visualized. An overview of the software development process in the method is illustrated in Figure 4.

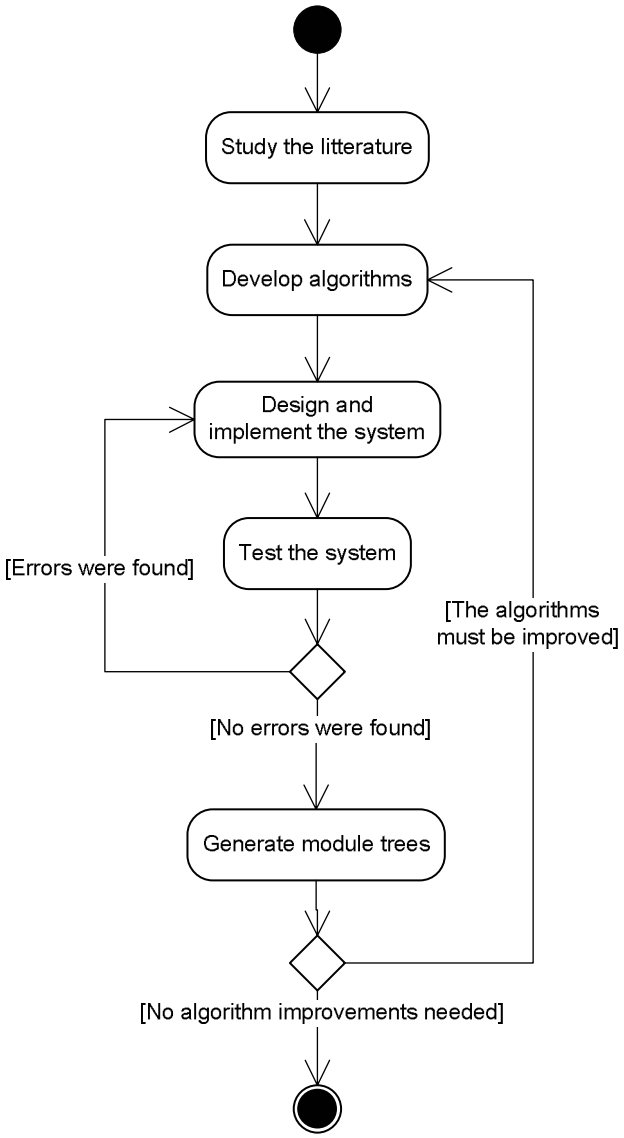

Figure 4: Overview of the iterative software development process.

First important concepts in bioinformatics and graph theory must be understood. It is also important to investigate if there are existing algorithms that can be used. When developing algorithms, models can be used together with pseudocode to simplify the procedure. When algorithms have been developed, they must be implemented. The

software design is important here, to gain high software quality. The implemented functions must then be tested to avoid incorrect results. In this case, the system units can be tested by using trace logs with information about state and actions. If errors are found, the implementation needs to be corrected. If no errors are found, test output will be generated to be able to analyze the quality of the results. The correctness of the whole system may be checked here by comparing the actual results to the expected. If the results are insufficient, the algorithms need to be refined. Otherwise, the software development is finished, and the program can be used to generate output that can be analyzed.

5 Algorithm development

In this chapter the algorithms for integration of semantic information with modular decomposition are described.

5.1 Developing algorithms

There are already several algorithms for graph decomposition that can be used to generate a modular decomposition of a protein interaction network. We need to assign functional information for the modules of a modular decomposition though.

To integrate semantic information about molecular function, semantic information has to be mapped in some way to modules. But before starting to develop new or modify existing algorithms, the type of annotation sources and datasets for protein interactions have to be chosen, because they may have different semantics. The protein interaction graphs used in this study are generated with yeast two-hybrid technology (Ito et al., 2001), and terms from Gene Ontology (GO) will be used to describe the biological of proteins. The protein interaction graphs and the annotations for the proteins are collected from the *Saccharomyces Genome Database* (SGD)[7]. SGD contains large datasets with information of interactions and GO-term annotations for proteins in the yeast *Saccharomyces cerevisia*.

5.1.1 Minimum subsumers

Because semantic information needs to be mapped to modules, the functional similarity of the proteins in a module is important. The *minimum subsumer* of a set of terms can be used to see how the proteins in a module are functionally related. Each protein has a set of GO-terms annotated and from these term sets the minimum subsumer of a module can be identified. Lord et al. (2003) defines the minimum subsumer of two terms as the shared ancestor term in the GO-term graph, with the minimum *probability*. The probability p for a GO-term is the probability that a randomly chosen protein can be classified with the specific GO-term. This probability can be obtained by counting the number of times a term occurs in a bioinformatic database and divide this with the total number of annotations in the database:

$$p(t) = \frac{freq(t)}{N} \tag{2}$$

A term occurs if it or any of its descendants in the GO-term graph occurs (Lord et al., 2003). The less specific a term is, the higher is the probability for the term. So, for example, the probability of the term “molecular function” is 1, the probability of the term “receptor”, which is a child term to “molecular function”, is approximately 0.124 and the probability of one of its child terms, the term “transmembrane receptor”, is approximately 0.0997. In Figure 5 the minimum subsumer concept is illustrated for the gene products BRR1 and SMX2 (a gene product is either RNA or protein).

[7] http://www.yeastgenome.org/

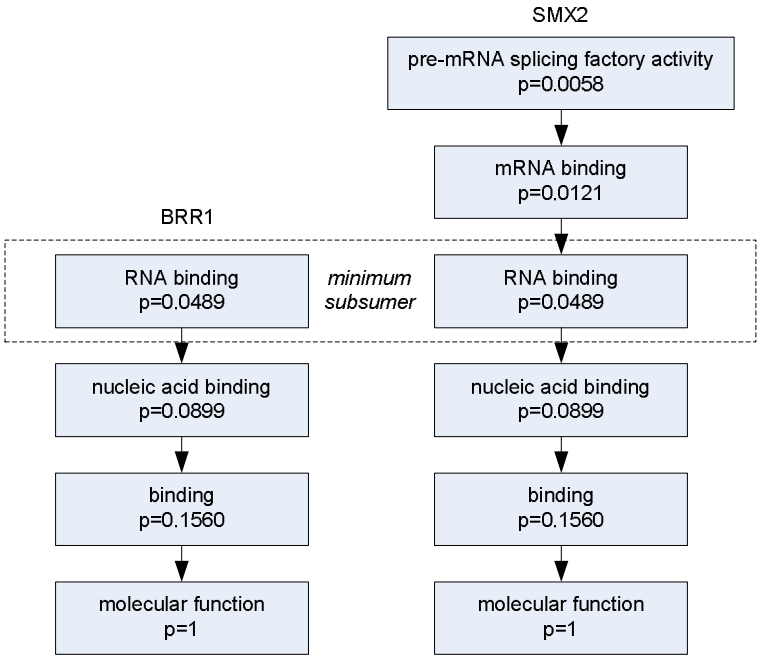

Figure 5: The minimum subsumer term of the gene products BRR1 and SMX2 is "RNA binding" (Lubovac et al., 2005).

5.1.2 The integration procedure

Two different approaches for the solution of the problem with integration of semantic information were considered. The first approach was to integrate the functional similarity into a modular decomposition algorithm and then map GO-terms to the modules. The second approach was to only map GO-terms to the modules created from an existing modular decomposition algorithm. Because the first approach seemed to be hard to implement and to analyze the plausibility of the output, the second approach was chosen. Although the modular decomposition algorithm was not changed, an algorithm for completing the modular decompositions with *semantic sub-modules,* created from functional similarities, was developed. With that algorithm it also became possible to create a kind of modular decomposition solely based on functional similarities between proteins.

Decompositions with no semantic sub-modules will be referred to as *graph decompositions*. Graph decompositions completed with semantic sub-modules will be referred to as *hybrid decompositions,* and the decompositions with only semantic modules will be referred to as *semantic decompositions.*

The mapping of semantic information to a modular decomposition was decided to build upon minimum subsumers for the leaf proteins of a module. When the minimum subsumer term of a module is determined, the module can be assigned information based on that term. With *cutoff-conditions,* modules whose proteins do not share functional similarity significantly can be cut from the tree when the result is visualized or exported. The most important algorithms used for the integration of semantic information to modular decompositions are described in the following sections.

5.2 Important algorithms

In this section algorithms that can be used to identify minimum subsumers of modules and algorithms for creation of semantic sub-modules are described in pseudocode.

5.2.1 Identifying minimum subsumer of a module

The purpose of the first two algorithms is to identify the minimum subsumer term of a tree. First the term set of each leaf protein is added to a list of term sets. Then the intersection of all sets is identified. The term set of a protein consists of all the annotated terms, including the ascendant terms in the GO-graph, for the protein. The resulting term is a term with the minimum probability.

```
function GetMinimumSubsumer (T: semantic module tree) : returns term
    L ← the leaf proteins of T.
    S ← {}
    For each protein p in L do
        s ← the set of terms annotated for p.
        S ← S ∪ s
    end.
    ms ← GetMinimumSubsumer( S ).
    return ms.
end.
```

Algorithm 1: Get minimum subsumer term of a tree.

```
function GetMinimumSubsumer (S: set of term sets) : returns term
    it ← an element of S.
    S ← S \ {it}
    For each set s in S do
        it ← it ∩ s
    end.
    ms ← the term "molecular function"
    For each term t in it do
        If probability (t) < probability (ms) then
            ms ← t
        end
    end.
    return ms.
end.
```

Algorithm 2: Get minimum subsumer term of a tree.

5.2.2 Semantic sub-modules

A semantic sub-module is a module created from the child proteins of the parent module, such that the sub-module has a more specific minimum subsumer than the parent. Moreover a semantic sub-module may not contain all child proteins of the parent module, but the module must contain at least two proteins.

More exactly a semantic sub-module *S* can be defined as a rooted child tree of an internal node *m* in a semantic module tree, which is created from the child proteins of *m* such that:

- The set of leaves in *S* is a proper subset of *m*'s children, and
- The minimum subsumer of *S* is less probable than the minimum subsumer of *m*, and
- *S* has at least two leaves, and
- It is not possible to form a semantic sub-module *T* of *m* with more leaves than *S* has.

5.2.3 Adding semantic sub-modules

There are two algorithms that are used to add semantic sub-modules to a tree. The first algorithm traverses a semantic module tree and uses the second algorithm to create sub-modules to the modules, if possible.

```
procedure AddSemanticSubmodules (M : semantic module tree)
    ms ← minimum subsumer of M.
    V ← vertices in M.
    For each v ∈ M do
        If v is a semantic module tree then
            AddSemanticSubmodules (v)
        End if
    End.
    CreateSemanticSubmodules (M, probability (ms) )
End.
```

Algorithm 3. Adding semantic sub-modules to a module tree.

In the second algorithm, semantic sub-modules are created from the semantic module tree M.

```
procedure CreateSemanticSubmodules (M: semantic module tree, prob_M: probability of M)
    fin ← false
    P ← child proteins of M
    Remove all child proteins of M.
    While P consists of more than 2 proteins and while fin is false, do
        S ← {}
        For each protein prot in P do
            s ← set of annotated terms for prot
            S ←S∪ {s}
        End.
        size_P ← number of elements in P.
        mct ← GetMostCommonTerm (S, prob_M, size_P )
        If mct is a term then
            P_2 ← the proper subset of proteins in P that have the term mct annotated
            P ← P \ P_2
            M: new semantic module tree
```

```
            AddSemanticSubmodules (M, P2, probability (mct) )
            Add M to the child set of M.
        else
            fin ← true
        End if.
    End
  Let the proteins that are left in P be children of T.
End.
```

Algorithm 4: Creating semantic sub-modules from the child proteins of a tree node.

5.2.4 Identifying the most common term for a set of proteins

There is an algorithm that identifies the most common term among the annotated terms for a set of proteins. The algorithm first identifies a term, that is shared by at least two protein's term sets The probability of this term should be below the threshold value and the term may not belong to more than *N*-1 term sets, where *N* is the number of child proteins of the parent.

```
function GetMostCommonTerm( S: set of term sets, th: threshold probability, max: upper limit for number of elements) : returns a term or null
  T: empty set of terms
  For each element s in S do
    For each term t in s do
      If counter[t] is set and probability (t) < th then
        counter[t] ← counter[t] + 1
      else
        T ← T ∪ { t }
        counter[t] ← 1
      end.
    end.
  end.
  greatest ← 0
  For each term t in T do
    if (counter[t] > greatest) and (counter[t] < max) and (counter[t] > 1) then
      mct ← t
      greatest ← counter[t]
    end
  end.
  return mct.
end.
```

Algorithm 5: Identifying the most common term with a probability below a certain limit and with a maximum number of term occurrences allowed.

6 Design and implementation

In this chapter, the design and the implementation of the system is described in detail.

6.1 Overview of the system

To be able to reuse as much source code as possible, the system was decided to extend Modulor. The system will assign minimum subsumer term information to the modules and also create semantic sub-modules. Even if it would have been most desirable if the visualization functionality would reside within the extended Modulor, a separate tool will be used for the visualization. The reason for this is that in this project, the emphasis is put on the decomposition procedure.

Both GraphViz and Pajek are freeware tools that can be used for visualization. Since GraphViz has a simple language for describing graphs, it was considered to be the best choice for this project. More output formats can be added later though.

Modular decompositions from the program will also be exported as data tables with additional statistic information. An overview for how Modulor will be extended is shown in Figure 6.

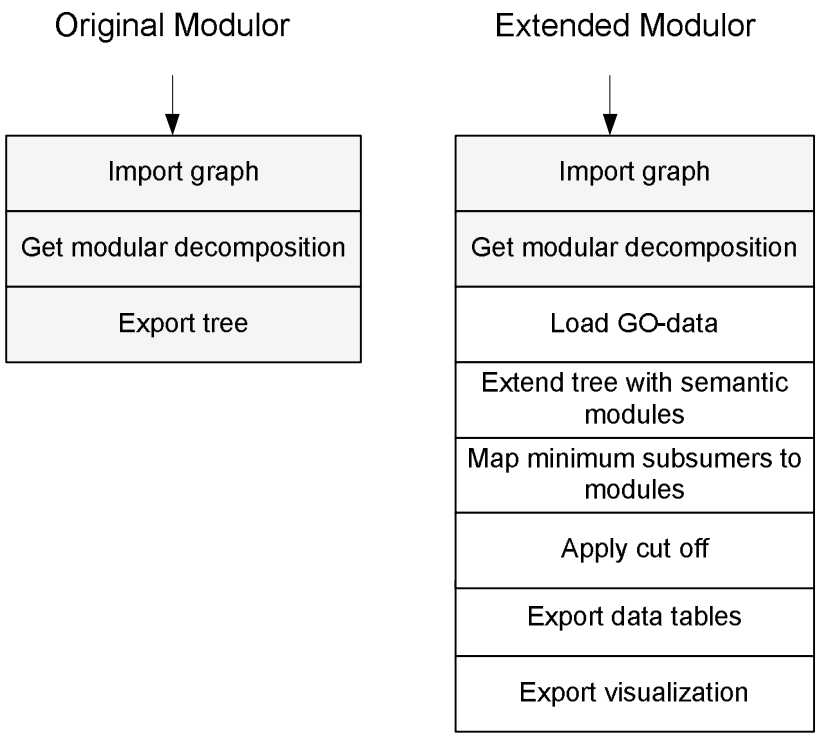

Figure 6: Overview of the extended Modulor system. The first steps in both versions are to import the graph from an xml-file and to create the modular decomposition.

6.2 User interface

The Java-application will run from the console only, because a GUI was not considered necessary at this stage. To produce output a user can write:

```
java –jar Modulor.jar xml_file
```

A directory with the name of the xml-file is created in the output directory (if it not exists already). In that directory all output-files for the graph are placed. For each decomposition method (graph, hybrid and semantic) three files can be generated.

Besides the GraphViz dot-file there is a file that describes the tree with its modules, and there is also a file with statistics for the GO-terms that are represented in the tree. The file `Modulor.ini` can be used to set paths for the GO-data input files and to set the cut off threshold value. If the cut off threshold value is set to 0, no sub-trees will be removed.

6.3 Existing implementation of Modulor

To be able to reuse the source code for modular decomposition in Modulor, the system will be programmed in the Java language. Modulor consists of the classes `Modulor`, `Graph`, `Edge`, `Vertex`, `GraphLoader`, `ModuleTree` and some classes that are used for the vertex partitioning procedure. `Modulor` is the main class, and from that the graph is imported from an xml-file by using `GraphLoader`. When the graph is loaded, the modular decomposition is determined by the `Graph` class and by some helper classes for the vertex partitioning. The modular decomposition is then stored in a `ModuleTree` object. When the module tree is determined, it is exported to an xml-file.

6.4 Design

A new class `SemanticModuleTree` was developed to create module trees with minimum subsumers assigned to the modules (these will be referred to as *semantic module trees*). This class needs a class `SemanticDataLoader` to load semantic information and store this information in objects of the classes `ProteinGO` and `TermGO`. A `ProteinGO` object holds information of a specific protein and objects of class `TermGO` holds information about specific GO-terms. A simplified class diagram is shown in Figure 7.

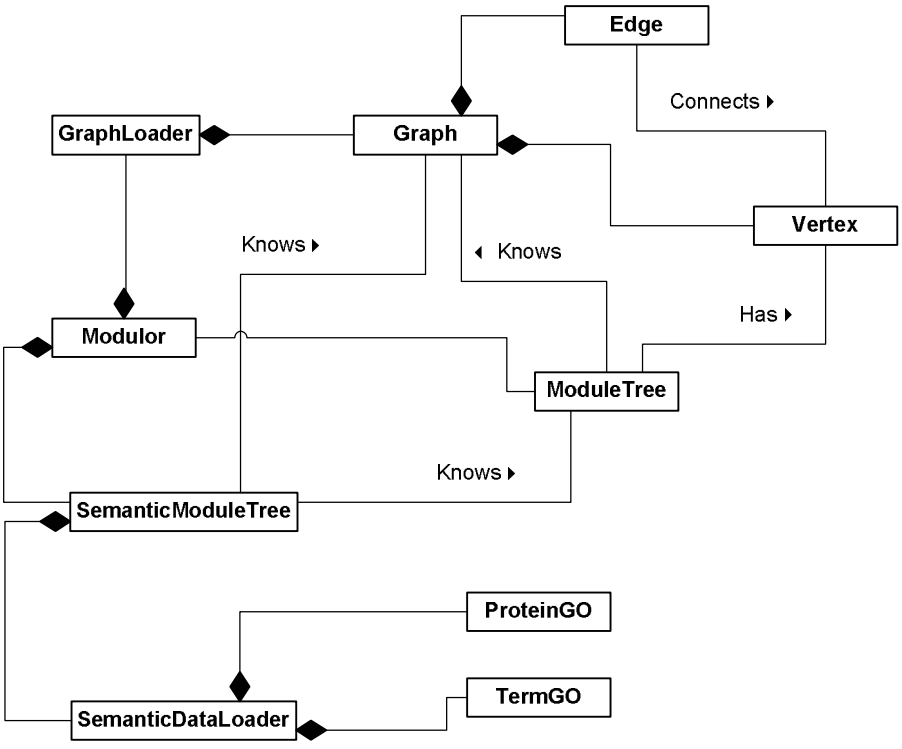

Figure 7: Overview of the classes used in the extended Modulor (the helper classes for the modular decomposition procedure are not shown).

6.4.1 The SemanticModuleTree class

The `SemanticModuleTree` class deals with creation of decompositions from `Graph` objects or `ModuleTree` objects. The class can create graph, semantic and hybrid decompositions. Graph decompositions are decompositions from the original Modulor program, with functional information presented for its modules. A semantic decomposition is a semantic module tree, with semantic sub-modules as internal vertices created from the protein set of a graph. The hybrid decomposition combines both methods, such that, for each module in a graph decomposition the class tries to create semantic sub-modules from the child proteins of the modules.

There are four different constructors for the `SemanticModuleTree` class:

- A constructor that takes a label name as parameter. This constructor creates a semantic module tree with one vertex: the labeled root.
- A constructor that takes a `ModuleTree` object as parameter. This constructor transforms a `ModuleTree` object into a `SemanticModuleTree` object.
- A constructor that takes a `Graph` object as parameter. This constructor creates a `SemanticModuleTree` root node with the proteins in a graph as children.
- A constructor that takes a `Collection` object, consisting of `ProteinGO` objects, as parameter. This constructor creates a `SemanticModuleTree` from a set of proteins.

There are some important methods for the `SemanticModuleTree` class, which are described below.

- `getMinimumSubsumer` – There are two functions, which have the name `getMinimumSubsumer`. The first takes no arguments and the second takes a list of terms as argument. These functions are used to identify the minimum subsumer of a module.
- `addSemanticSubmodules` – There are two recursive overloaded functions `addSemanticSubmodules`. The first takes a `SemanticModuleTree` object as an argument and the second takes a `SemanticModuleTree` object, a set of `ProteinGO` objects and a threshold p-value as arguments. These functions are used to add semantic sub-modules to a `SemanticModuleTree` object.
- `getMostCommonTerm` – This function takes a list of term sets, a threshold *p*-value and the maximum number of term sets that may contain a term. This function returns the term that is most commonly contained in the term sets. The term must also have a p-value (probability of the term) *below* the threshold value.
- `getProteins` – This function takes a set of proteins and a term as argument. The function gives the subset of proteins that have the specific term annotated.
- `applyCutOff` – This function takes a threshold p-value as arguments and its only purpose is to call the function `cutOffSubTrees`. The `cutOffSubTrees` function takes a `SemanticModuleTree` object and a threshold p-value as argument. The function removes sub-trees that only contain modules with a p-value above or equal to the threshold value.
- `exportModules` – This function is used to export the modules into a table format. The function also exports statistics for the modules.

- `exportTerms` – This function exports information about the GO-terms in a semantic module tree.
- `exportDot` – This function exports a semantic module tree into a dot-format, by calling an overloaded function with the same name.

6.4.2 The ProteinGO class

The purpose of the `ProteinGO` class is to store and handle data for a protein. It has getter functions for all its data, a function to check if the protein has a specific term annotated and a function that returns all annotated terms for the proteins.

6.4.3 The TermGO class

The class `TermGO` stores data for a GO-term. It has getter functions for all its member variables and a function that checks if the term is descendant to the term "molecular function".

6.4.4 The SemanticDataLoader class

The class `SemanticDataLoader` imports semantic information about proteins and GO-terms. This information is stored into `ProteinGO` objects and `TermGO` objects. The class also creates mappings from protein names to `ProteinGO` objects, from GO-term id's to `TermGO` and from `ProteinGO` objects to clustering coefficient values. The class also loads settings from the file Modulor.ini.

6.4.5 Changes to the Modulor class

Several changes are needed for the `Modulor` class. The GUI-code will be removed and the main function will create `SemanticModuleTree` objects with the three types of decompositions: graph, semantic and hybrid decompositions. The class will also apply cutoff (optional) and export the decompositions into a number of formats.

6.5 Implementation

In this section, the implementation of some important functions will be described.

6.5.1 Identify minimum subsumer for a module

In the function `getMinimumSubsumer()` a list `goTermList` is used to hold sets of terms for the proteins in the `SemanticModuleTree` object. When the list is created a while loop iterates through the set of leaf proteins and for each protein, the term set of the protein is added to the list:

```
public TermGO getMinimumSubsumer()
{
    LinkedList goTermList = new LinkedList();
    // get iterator for the leaves
    Iterator it = getLeaves().iterator();

    // for each leaf add the first annotated term to the list of
    // terms
    while (it.hasNext()) {
        ProteinGO prot = (ProteinGO) it.next();
        goTermList.add(prot.getAllTerms() );
    }
```

When the list of term sets is ready the resulting `TermGO` object of the call to the function `getMinimuSubsumer(LinkedList)` with the `goTermList` as argument is returned:

```
        // get minimum subsumer term
        TermGO ms = getMinimumSubsumer(goTermList);

        // return minimum subsumer term
        return ms;
    }
```

In the function `getMinimumSubsumer(LinkedList)` the term to return, `min`, is initialized to the term "molecular function", and the set intersection is initialized to `null`. An iterator `it` for the list `termList` is also retrieved. If the list consists of at least one element, `intersection` is assigned that set. The next step is to iterate over the rest of the elements in `termList` and to get the intersection with those elements and the set `intersection`.

```
    public TermGO getMinimumSubsumer(LinkedList termList)
    {
        TermGO min =  (TermGO) idToTermMap.get("GO:0003674");
        HashSet intersection = null;
        Iterator it = termList.iterator();

        // let the set intersection be the first set in the list
        if (it.hasNext())
        {
            intersection = new HashSet( (HashSet) it.next() );
        }

        // get intersections with the other sets
        while (it.hasNext() )
        {
            HashSet set = (HashSet) it.next();

            // get the intersection
            intersection.retainAll(set);
        }
```

Now `intersection` will hold the shared terms from the sets. The iterator `it` will now be used to iterate through the set `intersection`, in order to find the term with the minimum probability. That term will then be returned.

```
        if (intersection != null)
        {
            // find the term with minimum p-value
            it = intersection.iterator();
        }

        while (it.hasNext() )
        {
            TermGO term = (TermGO) it.next();
            if (term.getPValue().doubleValue()
                    < min.getPValue().doubleValue()  )
            {
                min = term;
            }
        }
```

```
        return min;
    }
```

6.5.2 Add semantic sub-modules to a module

In the method `addSemanticSubModules(SemanticModuleTree)` the minimum subsumer is first assigned to the variable `term`. Then the tree is traversed by adding its child proteins to the set `proteins` and by calling the function recursive for child trees.

```
    private void addSemanticSubModules(SemanticModuleTree mdtree)
    {
        TermGO term = mdtree.getMinimumSubsumer();
        Set proteins = new HashSet();

        for (Iterator it=mdtree.getChildren().iterator();
                it.hasNext(); ) {

            Object n = (Object) it.next();

            if (n instanceof ProteinGO) {
                proteins.add( (ProteinGO) n ) ;
            }
            else if (n instanceof SemanticModuleTree) {
                // call this function recursive
                addSemanticSubModules((SemanticModuleTree) n );

            }
        }
```

When the set proteins consists of the child proteins of the tree, the child proteins are removed from the tree. Then the function `addSemanticSubModules(SemanticModuleTree,Set,double)` is called with the tree itself, the set `proteins` and the probability of the minimum subsumer as arguments.

```
        mdtree.removeChildren(proteins);

        // create semantic sub-modules for the children
        addSemanticSubModules(mdtree, proteins,
                                term.getPValue().doubleValue());
    }
```

In the function `addSemanticSubModules(SemanticModuleTree,Set,double)` there is a loop that loops while the set `proteins` consists of at least three proteins and while `fin` is `false`.

```
    private void addSemanticSubModules(SemanticModuleTree mdtree, Set
                            proteins, double threshold) {

        boolean fin = false;

        while (!fin && (proteins.size() > 2) ) {
```

A list `termSetList` is used to store the term sets for the proteins in the set proteins.

```
            LinkedList termSetList = new LinkedList();

            for (Iterator it=proteins.iterator(); it.hasNext(); ) {
                HashSet temp = ((ProteinGO)it.next()).getAllTerms();
```

```
            termSetList.add(temp);
        }
```

Then the most common term with a probability less than `threshold` (the probability of the minimum subsumer term of the module) is assigned to `term`. The variable `term` is `null` if no such term was found.

```
        TermGO term = getMostCommonTerm(termSetList, threshold,
                                        proteins.size() - 1);
```

If a term was found the proteins with the specific term is assigned to the set `proteins2`. The proteins in `proteins2` are then removed from `proteins`, because they will be leaves to the sub-module instead. A new sub-module with the label “sem” is created and the function is called recursive to create sub-modules to the newly created sub-module. The sub tree is the added as a child of the tree. If no term was found above, the variable fin is set to true to stop the looping.

```
        if (term != null) {
            HashSet proteins2 = getProteins(proteins, term);

            // remove the proteins from the set
            proteins.removeAll(proteins2);

            SemanticModuleTree subTree =
                new SemanticModuleTree("sem");
            addSemanticSubModules(subTree, proteins2,
                                  term.getPValue().doubleValue());

            // add sub-tree to result
            mdtree.addChild( subTree );

        }
        else {
            fin = true;
        }
    }
```

The last step is to add the proteins that will not be in any sub-module to the tree.

```
    // add all proteins that are left
    for (Iterator it=proteins.iterator(); it.hasNext(); ) {
        ProteinGO tempProt = (ProteinGO) it.next();
        mdtree.addChild(tempProt);
    }

}
```

The function `getMostCommonTerm(LinkedList,double,int)` was called from the previous function. It is used to find the most common term with a probability less than `threshold`. The term can be member of less than `maxNum` sets. If no such term is found, the function returns `null`. First the return variable is initialized to null and a map `numberMap` is created to map terms to the number of times the term occurs.

```
TermGO getMostCommonTerm(LinkedList termSetList, double threshold,
                                         int maxNum) {

    TermGO mct = null;
```

```
HashMap numberMap = new HashMap();
```

For each set in `termSetList` terms are checked to see if the probability is less than threshold.

```
// count the number of times each term occurs
for (Iterator it=termSetList.iterator(); it.hasNext(); ) {
    HashSet set = (HashSet) it.next();
    for (Iterator it2=set.iterator(); it2.hasNext(); ) {
        TermGO term = (TermGO) it2.next();

        // get significant difference
        if (term.getPValue().doubleValue()
                < threshold) {
```

Then the term is added as an entry in the `numberMap` if it not exists earlier and the initial value is 1. Otherwise the value is increased by 1.

```
                if (numberMap.containsKey(term)) {
                    int newnum = ((Integer)
                            numberMap.get(term)).intValue() + 1;
                    numbermap.put(term, new Integer(newnum) );
                }
                else {
                    numberMap.put(term, new Integer(1) );
                }
            }
        }
    }
```

When the `numberMap` is ready, a loop checks which term is most common, but not member of all sets in `termSetList`. At last, the term is returned (if there was any).

```
    // find the most common term
    Set keyset = numberMap.keySet();
    int greatest = 0;

    for (Iterator it=keyset.iterator(); it.hasNext(); ) {
        TermGO term = (TermGO) it.next();
        int num = ((Integer) numberMap.get(term)).intValue();

        if (num > greatest && num <= maxNum && num >1 )
        {
            mct = term;
            greatest = num;
        }
    }
    return mct;
}
```

7 Results

In this chapter the results are presented and discussed.

7.1 Input data

The input to the program will be different subsets of the protein interaction network of the yeast *Saccharomyces cerevisiae*. The subsets, or complexes, that will be used as input are the *OST* complex, the *LSM* complex and the *ORC* complex.

Knauer and Lehle (1998) states that the OST complex consists of all known subunits for the catalyzation of the key step of N-glycosulation, which is an essential protein modification. Because this complex is known to have a specific functionality, it can be used to show how biologically plausible modular decomposition is. The OST1 protein will be used as *central node* for the decompositions. This means that the node and its neighbors will be the nodes in the sub-graph.

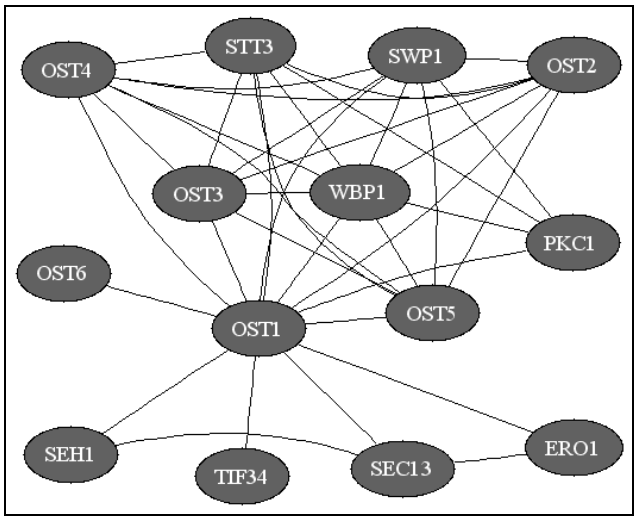

Figure 8: The OST complex with OST1 as central node.

OST2 will also be used as central node to see which impact the choice of central node has on the modular decompositions.

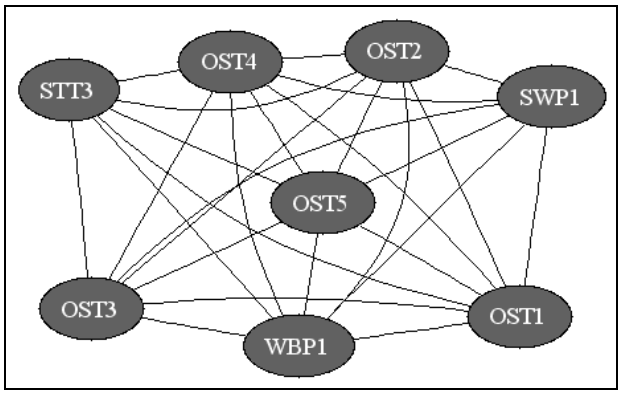

Figure 9: The OST complex with OST2 as central node.

The LSM complex has been extensively studied, and has known functions related to RNA processing (Dunn et al., 2004). This complex is also interesting because LSM6 interacts with all of the other proteins in the complex. The protein LSM1 and its

neighbors will form the protein interaction graphs.

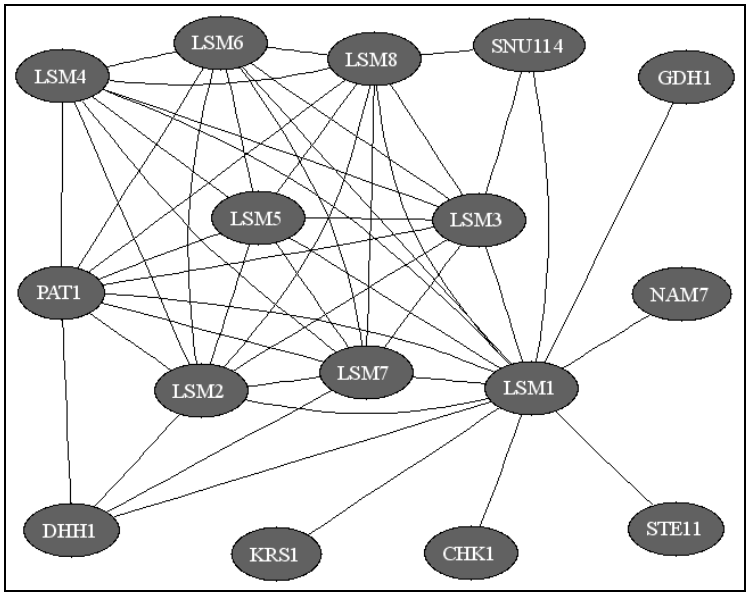

Figure 10: The LSM complex with LSM1 as central node.

Many studies have shown that the ORC complex directs all initiation of the eukaryotic DNA replication (Bell, 2002). Because of this, modular decompositions of this complex may give good coherence to functional similarity. The protein ORC1 and its neighbors will form the protein interaction graphs.

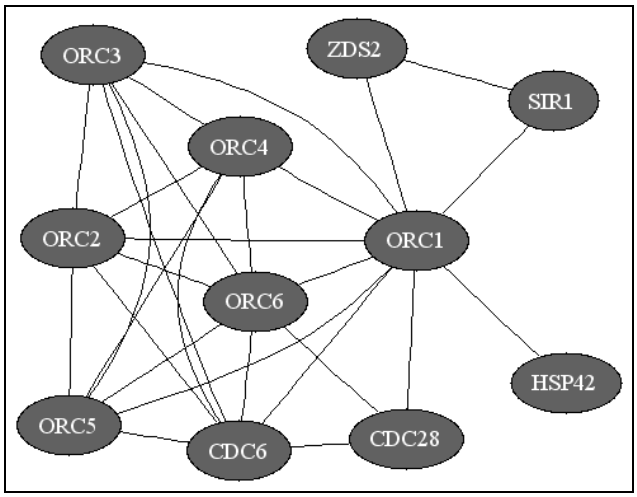

Figure 11: The ORC complex with ORC1 as central node.

7.2 Resulting trees

Graph, hybrid and semantic decompositions have been generated from the complexes OST, LSM and ORC. The trees will be presented with all modules (no modules have been removed).

The graph and hybrid decompositions of the OST complex give identical module trees. This fact, and that the probabilities are lower closer to the leaves, indicate that these decompositions are biologically plausible.

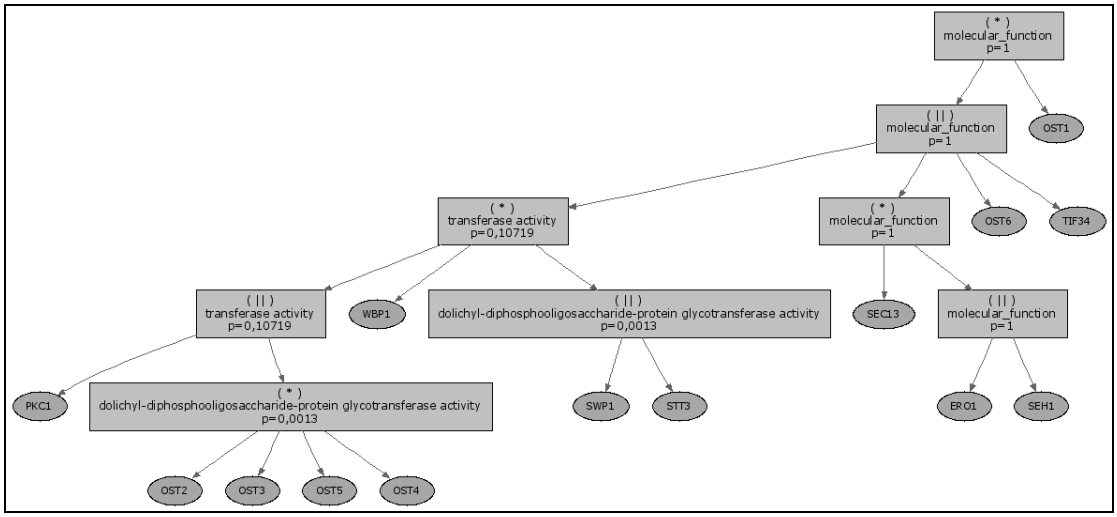

Figure 12: The graph and hybrid decompositions of the OST complex (with OST1 as central node) are identical. The modules are the rectangles and the proteins the ellipses. In the modules, the labels are within parentheses and below them are the functional description and the probability.

The semantic decomposition gives more proteins with more specific functional information, and the terms occurring for its modules also occur in the graph and hybrid decompositions. Even though there are more proteins in modules with a low probability, the structural information of the protein interaction graph is lost. The similarities of the decompositions indicate that the graph decomposition is biologically plausible for the OST complex. The fact that strongly functionally related proteins often reside in the same modules also an indication of this. One problem with the graph decomposition is that for example OST1 is placed in the series module that are root of the tree, and therefore it is hard to see how this protein is functionally related to the others. In the semantic decomposition the functionality of this cluster is clear though.

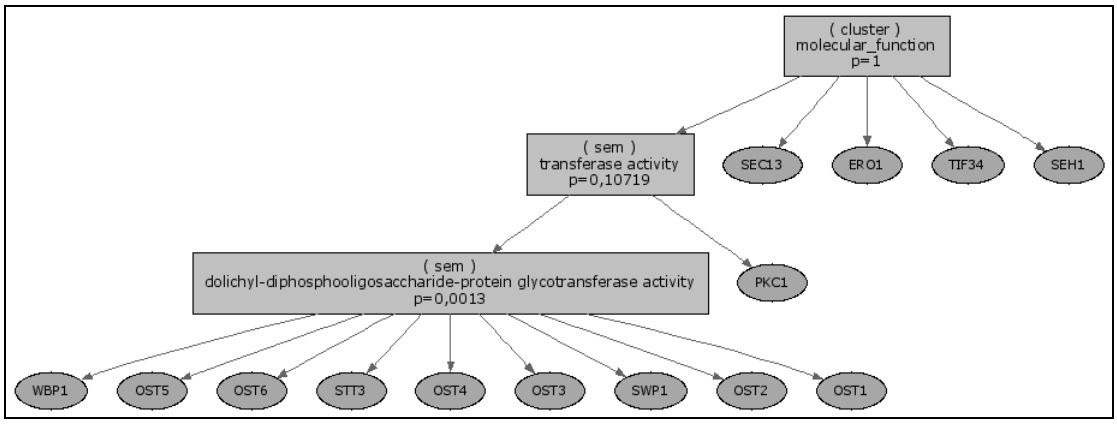

Figure 13: The semantic decomposition of the OST complex with OST1 as central node.

In the previous decompositions OST1 has been used as central node. When OST2 is used as central node instead, the decompositions became totally different. In the graph and hybrid decompositions, the proteins OST1-5 are children of a series node with the most specific term of the previous decompositions. The proteins SWP1 and STT3 are belonging to a child module of the root that has the same term.

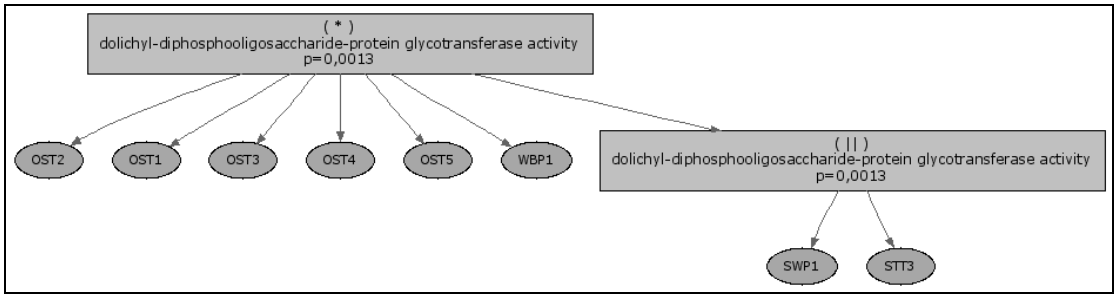

Figure 14: The graph and hybrid decompositions of the OST complex with OST2 as central node, are identical.

In the semantic decomposition there is only on module, with the same term as for the other decompositions. In this tree the functional similarities are as clear as with the graph and semantic decompositions, but the structural data of the protein interaction network is missed. Therefore the graph and hybrid decompositions are more informative in this case.

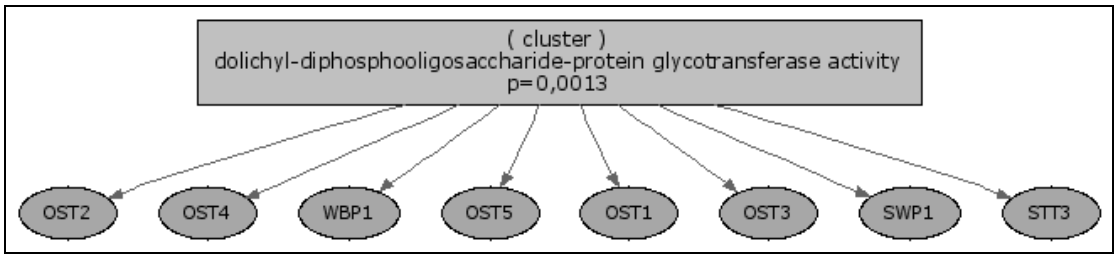

Figure 15: The semantic decomposition with OST2 as central node.

In the graph decomposition of the protein interaction graph of the LSM complex, there are two modules with a probability below 1. If the graph decomposition reflects functional relations between proteins, there should be more modules with low probability. But another choice of central node may give more specific terms for the modules in the graph decomposition.

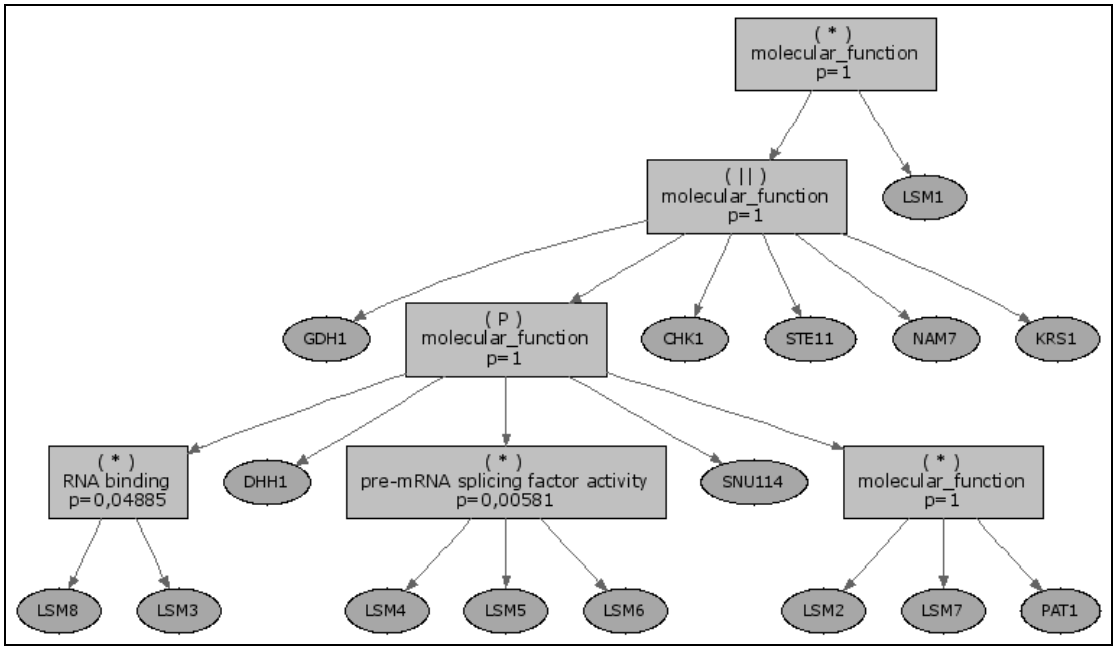

Figure 16: The graph decomposition of the LSM complex.

In the hybrid decomposition of the LSM complex, two semantic sub-modules are added from modules with the minimum subsumer “molecular function”. First there is a module with the term “RNA binding” that is created from the proteins LSM7 and LSM2. The term of this module indicates that also these proteins should be a part of the known functionality of the LSM complex – RNA processing. Then there is also a module with the term “protein kinase activity”.

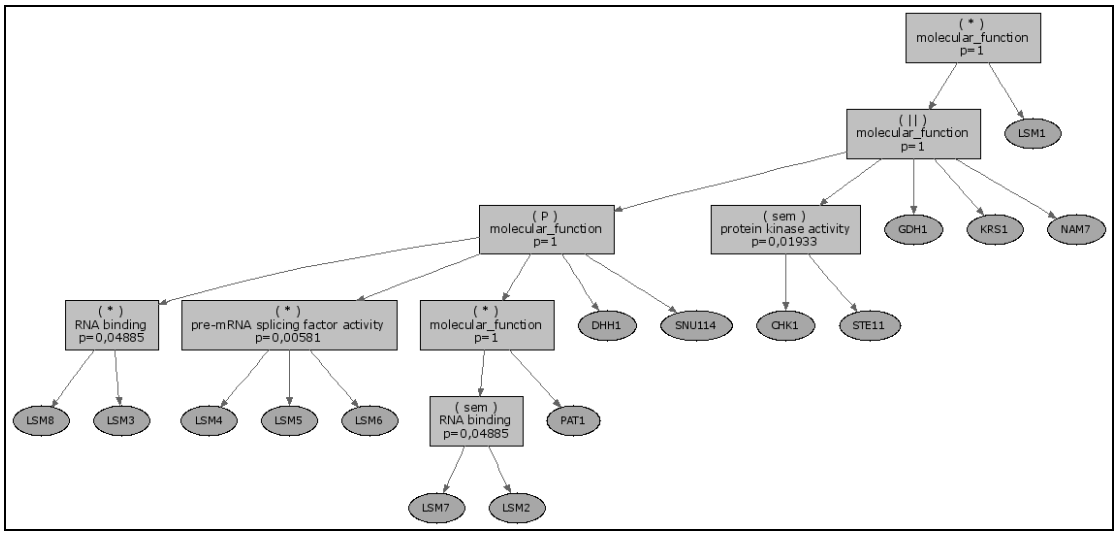

Figure 17: The hybrid decomposition of the LSM complex.

In the semantic decomposition of the LSM complex, the RNA processing functionality is much more clear than with the other decomposition methods. Therefore, the semantic decomposition might give a better idea of how the proteins are cooperating in this complex.

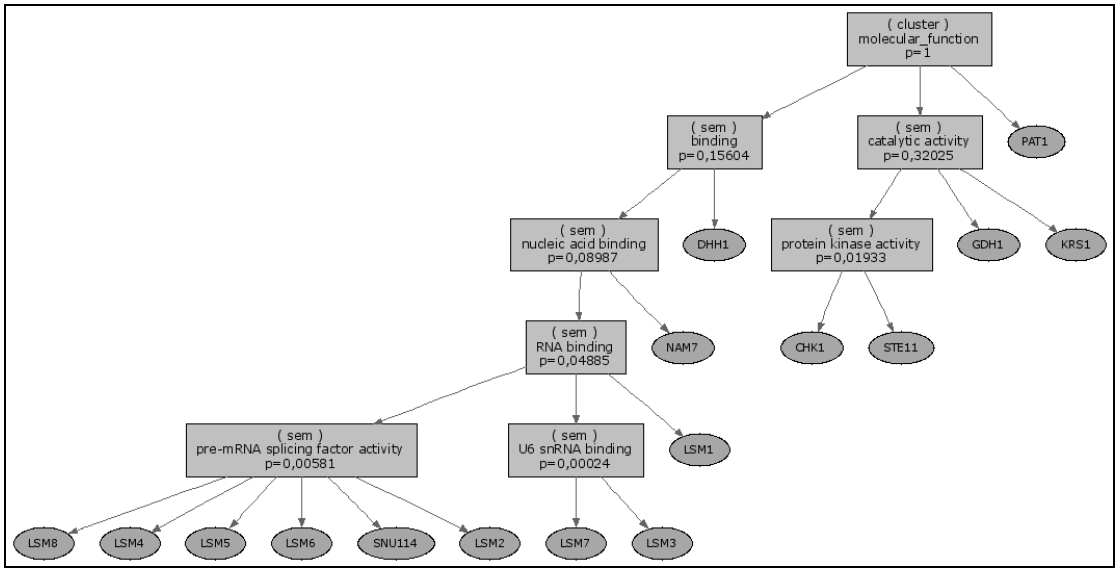

Figure 18: The semantic decomposition of the LSM complex.

The graph decomposition of the ORC complex does not give much coherence between the decomposition and the known functionality of the complex (DNA replication), except for the proteins ORC2, ORC3, ORC4 and ORC5.

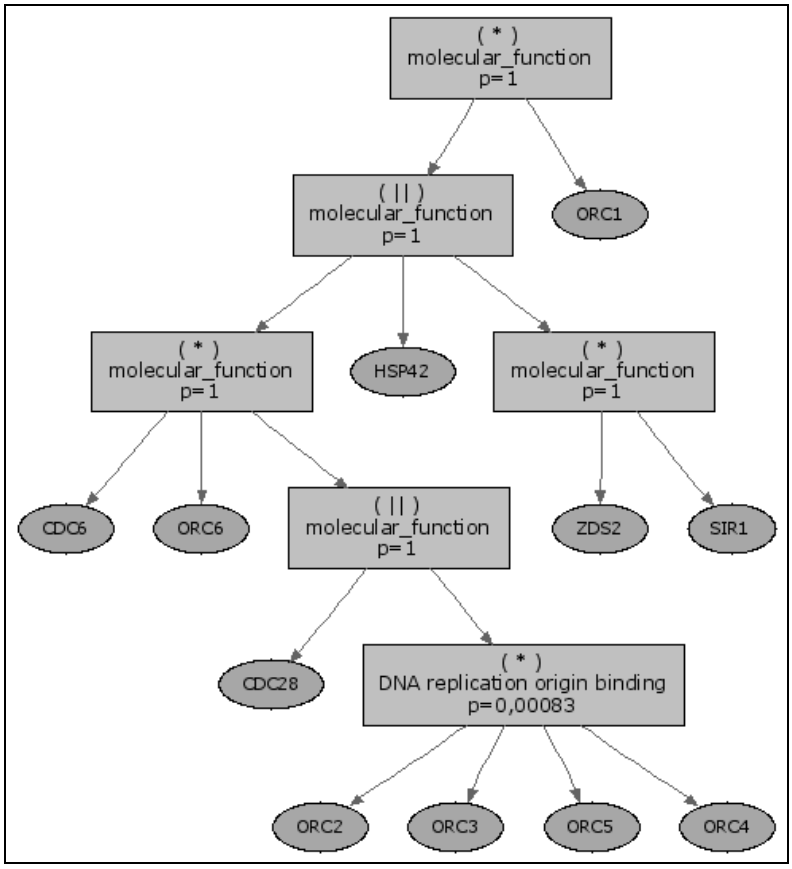

Figure 19: The graph and hybrid decompositions of the ORC complex (with ORC1 as central node) are identical.

With a semantic decomposition, many functional similarities are shown that are not obvious in the graph and hybrid decompositions. The coherence between the known function of the cluster, DNA replication, and the modules is very clear in the semantic decomposition. In this case the semantic decomposition might be more informative because of that. The lack of structural information in the decomposition is off course negative though.

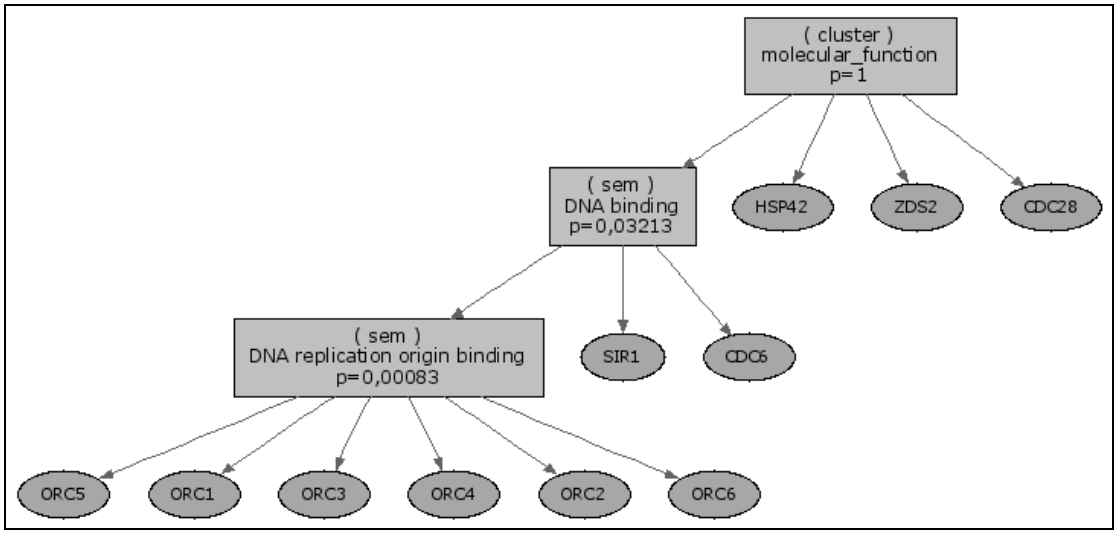

Figure 20: The semantic decomposition of the ORC complex.

8 Conclusion

In this chapter the results are summarized. Some future work is also proposed.

8.1 Summary

The coherence between modular decompositions (graph decompositions) and functional similarities is rather good for the complexes OST, LSM and ORC. The structure of the module tree from graph decomposition may, to a part, hide functional relationships though, because proteins do not always belong directly to modules with similar functionality. The choice of central node is important, because the coherence between the modules and the functional similarity of their proteins can depend much on this. In some cases, the modular decomposition can better show functional relationships if semantic modules are created from the original modules. These modules consist of functionally related proteins. Modular decompositions based solely on functional similarities are sometimes better to show the functionality of complexes and which proteins might cooperate to perform certain biological functions. It is not possible to decide which decomposition method is the best though, but by analyzing the different decompositions together, new module patterns should be easier to found than with only graph based modular decomposition.

The results show that many modules in a graph decomposition consist of proteins with similar function. This indicates that modular decomposition might give biologically plausible modules, but to be sure of this, much more clusters have to be studied. Statistical measures are also needed to be able to be confident of the plausibility of modular decomposition of protein interaction networks.

Minimum subsumers of modules are a good way of describing shared functionality of proteins in a module. Sometimes there can be several interesting terms that describe a set of proteins though, which is not easy to visualize. But presenting the modules in modular decompositions together with their minimum subsumers is much more informative than without them. The minimum subsumers can help to verify the biological plausibility of decompositions by showing which proteins are functionally related according to annotations.

8.2 Discussion

By assigning GO-terms to modules in a modular decomposition, the functional relations between the proteins become clear. This mapping of semantics to modular decomposition might also be used in other areas where graphs are used. To be able to do this, some kind of annotations is required for the objects represented by the nodes. No underlying graphs are needed with semantic decompositions, though. The usage of semantic decomposition might be useful for semantic objects, like words in a text. With a decomposition of a text, similar words can be grouped together, which might be useful to find the main subjects in the text, for example.

Software that maps minimum subsumers to a modular decomposition may be useful in the analysis of modular decompositions. The information about cellular functions of modules can be used for further experiments, where the proteins in a module with a certain function are being knocked out, for example.

8.3 Future work

There are many improvements of the system that can be done. A graphical user interface would be easier for the users – the user would be able to change preferences while the system is running and see the resulting graphs directly. The performance can be improved by optimizing algorithms and by holding semantic information in main memory between decompositions. For small clusters, the loading and parsing of semantic data is the most time-consuming process. For larger graphs, the algorithm performance may be a problem though. The current algorithms also handle some special cases arbitrarily, which is not appropriate. If two or more terms in the set of shared terms of a set of proteins have the same probability, the minimum subsumer can be any of the terms. When the most common term is being identified, there can be several candidates, which are equally common. In this case the term choice is arbitrarily.

The biological plausibility of different decomposition methods should be studied more deeply by analyzing more complexes. For graph decompositions the amount of specific modules can be used for this. Modular decompositions of a cluster in a protein interaction network can be compared to a cluster composed of the proteins in the same cluster where the interactions are randomly generated. If the decompositions created from real interaction networks are significantly better, modules in a modular decomposition should have biological plausibility. To verify the plausibility of the modules in semantic and modular decompositions, information about protein locations can be used. If proteins in a module are physically separated, the plausibility can be questioned. Protein interaction data can also be used for verification of semantic modules – if no interactions have been found between two proteins that reside in the same semantic module, the module is not plausible.

References

Barabási, A-L., Oltvai, Z.N.: Network biology: understanding the cell's functional organization. *Genetics*, Vol. 5, (2004) 101-113.

Bell, S. P.: The origin recognition complex: from simple origins to complex functions. *Genes & Development*, Vol. 16, No. 6, (2002) 659-672.

Deng, M., Sun, F., Chen, T.: Assessment of the reliability of protein-protein interactions and protein function prediction. *Pacific Symposium on Biocomputing*, 8:140-151(2003).

Dunn, R., Dudbridge, F., Sanderson, C. M.: The Use of Edge-Betweenness Clustering to Investigate Biological Function in Protein Interaction Networks. *BMC Bioinformatics*, Vol 6:39, (2004).

Gagneur, J., Krause, R., Bouwmeester, T., Casari, G.: Modular decomposition of protein-protein interaction networks. *Genome Biology*, Vol. 5:R57, (2004).

Habib, M., de Montgolfier, F., Paul, C.: A simple linear-time modular decomposition algorithm. *SWAT 2004, 9th Scandinavian Workshop on Algorithm Theory*, (2004) .

Hartwell, L.H., Hopfield, J.J., Leibler, S., Murray, A.W.: From molecular to modular cell biology. *Nature*, Vol. 402, (1999) 47-52.

Ito, T., Chiba, T., Ozawa, R., Yoshida, M., Hattori, M., Sakaki, Y.: A comprehensive two-hybrid analysis to explore the yeast protein interactome. *PNAS*, Vol 98, (2001) 4569-4574.

Knauer, R., Lehle, L.: The Oligosaccharyltransferase Complex from *Saccharomyces cerevisiae*. *The Journal of Biological Chemistry*, Vol 274, (1998) 17249–17256.

Lord, P.W., Stevens, R.D., Brass, A., Goble, C.A.: Investigating semantic similarity measures across the gene ontology: the relationship between sequence and annotation. *Bioinformatics*, Vol. 19, (2003) 1275-1283.

Lubovac, Z., Gamalielsson, J., Olsson, B. and Lindlöf, A.: Exploring protein networks with a semantic similarity measure across Gene Ontology. *Proceedings of the 6th International Symposium on Computational Biology and Genome Informatics*, USA, (2005).

McConnell, R.M., Spinrad, J.P.: Ordered vertex partitioning. *Discrete Mathematics and Theoretical Computer Science*, Vol. 4, (2000) 45-60.

Piehler, J.: New methodologies for measuring protein interactions in vivo and in vitro. *Curr. Opin. Struct. Biol.*, Vol. 15, (2005) 4-14.

Resnik, P.: Semantic Similarity in a Taxonomy: An Information-Based Measure and its Application to Problems of Ambiguity in Natural Language. *Journal of Artificial Intelligence Research*, Vol. 11, (1999) 95-130.

The Gene Ontology Consortium: Creating the gene ontology resource: design and implementation. *Genome Res.*, Vol 11, (2001) 1425-1433.

Zhong, J., Zhang, H., Stanyon, C. A., Tromp, G., Finley, R. L., Jr.: A Strategy for Constructing Large Protein Interaction Maps Using the Yeast Two-Hybrid System: Regulated Expression Arrays and Two-Phase Mating. *Genome Res.*, Vol. 13, (2003) 2691-2699.