Bioinformatics Leipzig
Institute of Computer Science
University of Leipzig

`orthoDeprime`: A Tool for Heuristic Cograph Editing on Estimated Orthology Graphs

Bachelor's Thesis

submitted by

Felix Kühnl

Advisors:

Dr. Maribel Hernandez-Rosales
Prof. Dr. Peter F. Stadler

February 2014

Thanks to Peter F. Stadler for pointing me to this interesting topic and for his straightforward help. Maribel, thanks for your great support and your time whenever I needed it. I also want to thank you, Sarah, for your help, ideas, discussions and software as well as for your friendship. Marcus Lechner and Sebastian Simoleit, thank you for your assistance with your tools. A special thanks goes to Werner Reutter and Marco Neumann for bravely paving the way through the hell of bureaucracy for the students of our faculty. Thanks to the chief scheduler, Petra Pregel.

Finally, I want to thank all my loved ones for supporting me: my girlfriend Saskia, my parents and my grandparents. Without you I would not be where I am today.

Abstract: It is a common task in modern biology to analyze or reconstruct the relationship of different species. Since it is assumed that related species share a common ancestor species, many genes are shared between those. Two genes from different species are called *orthologous*, if they originate from the same gene in their common ancestor species. An *orthology graph* on a set of genes X is the graph with a set of nodes X where any two nodes are connected if and only if they represent orthologous genes.

Recently, it has been discovered that a valid orthology graph is a cograph . However, the true orthology relation for X is unknown in practical applications, so it has to be estimated with an orthology detection tool. An example of such a program is `POFF` which uses sequence similarity and synteny information to determine orthologous genes in a given set of sequences. Unfortunately, errors are introduced by this estimation such that the orthology graph constructed from the output of `POFF` will in general not be a cograph.

This work focuses on the task of modifying estimated orthology graphs by adding and removing edges in a way that it becomes a cograph. This problem is known as *cograph editing* and is in general NP-complete, which is why a heuristic approach is chosen.

The motivation of the cograph editing process lies in the fact that it is possible to reconstruct the evolutionary history of the input genes when the cograph structure is restored. This information can then be used to reconstruct the phylogeny of a set of species. Another benefit is a more accurate orthology prediction resulting from the fact that the edited orthology graph will be more similar to the true one.

Selbstständigkeitserklärung: „Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann.“ [Fak14]

Leipzig, den _____ _____

Felix Kühnl

Contents

1 Introduction

As we know today, all organisms living on our planet are derived from a single or very few different simple organisms. Over millions of years, complex live forms as we know them evolved and each adapted perfectly to their habitat. It is surprising how closely related species are that at a first glance do not share any properties. Since Charles Darwin proposed the theory that all species are related and share a common ancestor, biologists try to study and to reconstruct the history of life on our planet. Figure 1 shows the result of an historic attempt to do so. Even in modern biology, equipped with knowledge and techniques that scientists of past centuries or just decades could not have dreamed of, the reconstruction of the relationships of species remains a challenging task.

Since today we are able to sequence the genetic code, we can analyze relationships of species directly at the level of genes. This work relies on the identification of *orthologous* genes to unveil the relationships of genes and even the relationships of entire species. Two genes from different species are called orthologous, if they originate from the same gene in their common ancestor species. It will be shown that one can interpret orthology as a mathematical relation with many properties that will be very useful to restore the desired information.

The orthology relation can be expressed as a graph that represents genes as vertices and connects exactly the orthologous vertices by an edge. For a valid orthology relation, this graph will be a *cograph* [Hel+13]. Cographs are a well-studied class of graphs. Using the knowledge of cographs, the relationship of the genes can finally be restored.

Since in practical applications for a set X of genes the true orthology relation is unknown, is has to be inferred approximately by a sequence comparison. A tool that does this job is `POFF`, which is used to generate the input data used in this work. When constructing an orthology graph from the output of such tools, however, one recognizes that their predictions are not always correct, and so the result will in general not be a cograph. The goal of this work is to modify the estimated orthology graph in a way that it becomes a cograph. This problem is formally known as *cograph editing* and is in general NP-complete [Liu+11], and thus it is hard to give an exact solution to this problem for graphs as huge as orthology graphs. This is why in this work a heuristic approach is chosen.

There are two reasons why cograph editing is an interesting task. Firstly, as already mentioned, from the cograph the relationship of genes and species can be reconstructed easily. Secondly, since the true orthology relation for the given set of genes X *must* be a cograph, editing the estimated orthology graph such that it becomes a cograph can also increase the precision of the orthology estimation.

This work is organized in seven chapters as follows. The first chapter, hopefully, introduced the reader to the topic, the goal and the motivation of this work. Chapter 2 provides the biological background information for readers not familiar with biology at all and defines orthology of genes. In chapter 3 the mathematical concepts required to model and process the biological input data are described. This includes

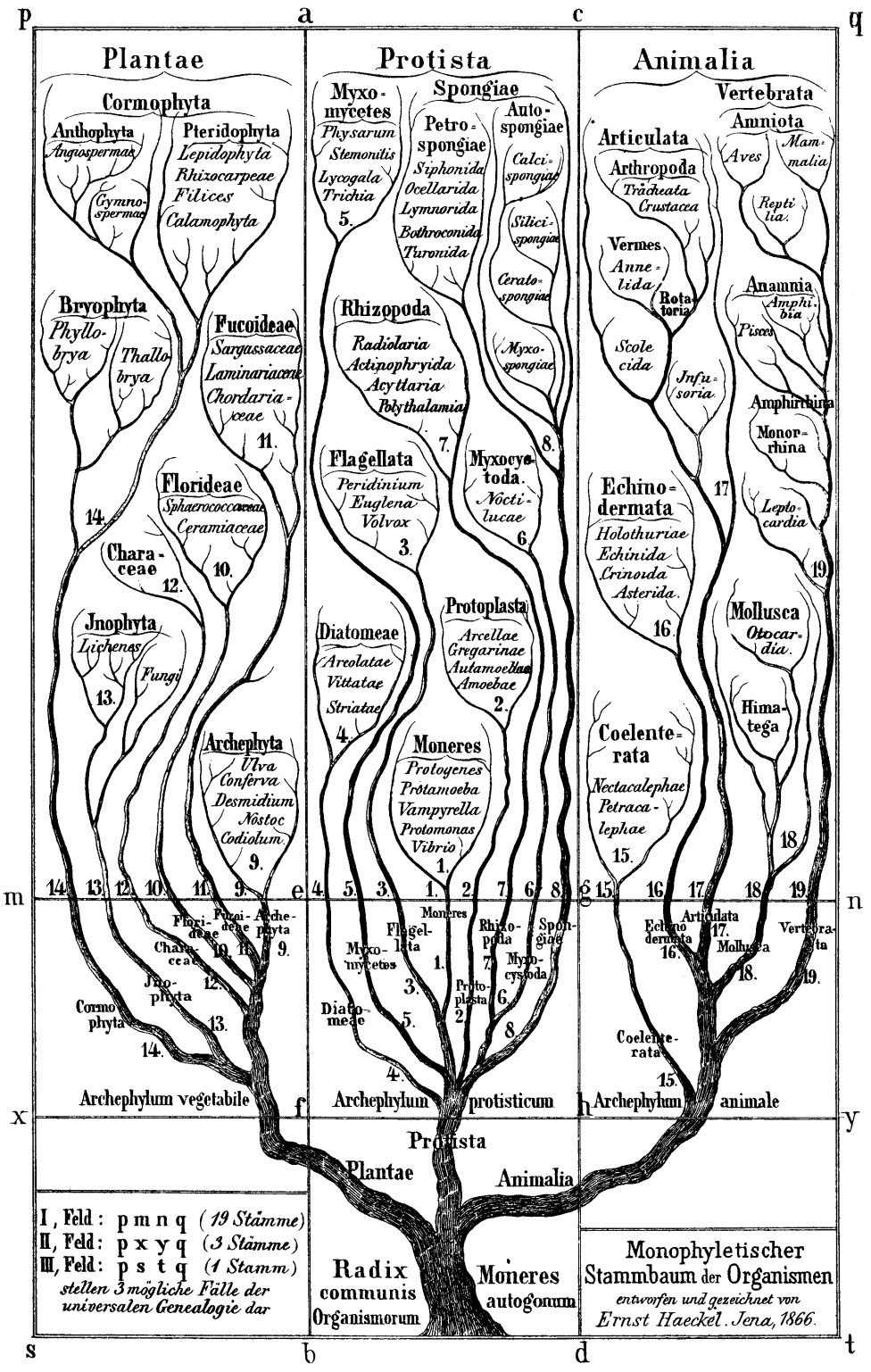


Figure 1: The *tree of life* [Hae66], hand-drawn by Ernst Haeckel in 1866. It is a historic document of the efforts made to reveal the relations of the species living on this planet.

an introduction to graph theory in general as well as a special focus on trees and cographs. It is explained why those notions are useful for solving biological problems. The next chapter, chapter 4, lists and explains tools and algorithms of other authors which are used during this work. Especially `POFF` is explained in detail, because its output is the foundation on which this work is build. In chapter 5 the prior information is combined to craft a heuristic tool called `orthoDeprime` which can transform graphs constructed from an estimated orthology relation to cographs. This program has been implemented in *C++* and is tested on simulated sequence data in section 6. Finally, section 7 concludes the results of this work.

2 Biological Background

Here, basic biological terms and concepts used in this work will be explained. The given information provides a background for the understanding of the tasks performed in the following chapters.

2.1 DNA and the Genetic Code

A basic property of any life form is that it reproduces itself. This reproduction process requires exact information on how to "construct" a "copy" of the existing organism. In all organisms that we know, this information is stored as *DNA (desoxyribonucleic acid)* in each of its cells. The DNA consists of sequences of pairs of the bases *adenine* (A), *cytosine* (C), *guanine* (G) and *thymine* (T), where A always pairs with T and G with C. The base pairs are connected by an acidic sugar-phosphate backbone and form a helical structure consisting of two strands. Due to the specific pairing of bases, each one of the two strands contains the entire genetic information.

Three base pairs form a so-called *triplet*, which encodes for a specific amino acid. This way a sequence of DNA determines a sequence of amino acids which is used by a cell to generate a *protein*, a molecule consisting of many connected amino acids folded to a certain structure. The different proteins fulfill numerous important functions in every cell. The association of the base triplets with their respective amino acids is known as the *genetic code* (summarized in figure 2) and is, with a few exceptions (see e. g. [WS08]), the same for all organisms we know. The latter property is referred to as the *universality* of the genetic code.

2.2 The Genome and Single Genes

Even in complex organisms with different specialized tissues, each cell contains the entire genetic information of that organism. It is called its *genome*. Not all of the DNA of a cell does indeed encode the information for a certain protein. Large parts of the genome seemed to have no function at all and had been referred to as "junk", though in the last years there has been found evidence [The+12] that those regions might have an important regulatory roll. A single sequence of DNA that is translated and synthesized to a protein by a cell is called a *gene*.

Note that, though the genetic code is uniquely determined, it is possible that from a single sequence of DNA different proteins can be synthesized due to the post-processing steps (e. g. *multiple splicing*) performed by the cell after the transcription process. Also, the genetic code is *degenerated*, which means that different triplets may encode for the same amino acid, and thus different genes can translate into the same protein.

During the course of this work, genes will be compared to each other. Since genes are sequences of base pairs, these sequences could be compared to each other directly. But since one is usually interested in the function of a gene, i. e. the function of the

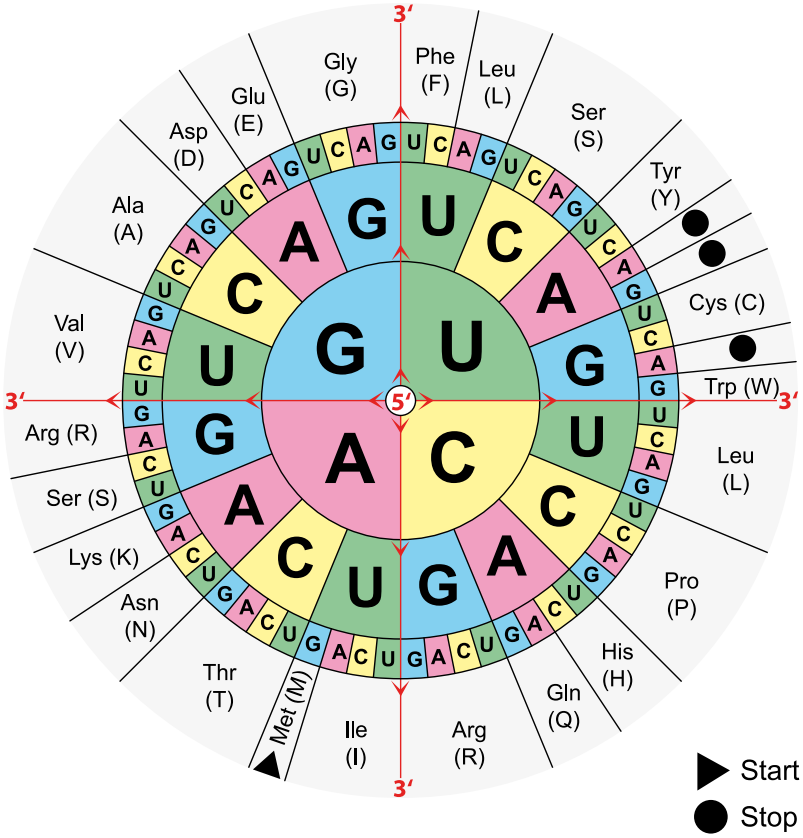


Figure 2: The genetic code [Mou14]. Starting in the center and following the letters of a triplet to the outside gives the amino acid this triplet encodes for. The amino acids are abbreviated by their respective three letter and one letter codes. Note that here T (Thymine) has been replaced by U (Uracil), since this figure shows the translation of *RNA* triplets to amino acids. RNA is the transcribed form of DNA.

protein it encodes for, it is often more useful to compare the sequence of amino acids in the encoded protein than the base sequence of the genes itself.

2.3 The Evolution of Life

On our planet we can see a vast variety of organisms and species. By today it is generally accepted, at least among scientists, that this variety arose by an evolutionary process. During replication, the genome of an organism is likely to change, or *mutate*. Most of those mutations are either repaired by the cell or do not have any effect at all, since they do not affect any gene or have no effect on the synthesized protein. Some however will change the properties of the organism, which can have positive or negative results for it. Here, a second mechanism comes into action: a *successful* organism is more likely to survive and to reproduce itself. This causes a *natural selection* of those and can cause a property invented by mutation to spread across an

entire *population*, the sum of the individuals of a species living in a certain geographic region.

A *speciation*, the process when a new species arises, is the result of the evolution of an already existing one. Today it is assumed that all *extant* species, i. e. the ones that have not become extinct, evolved from a single "primitive" organism over a time span of millions of years. As a consequence, all species are related to some point. The reconstruction of this complex relations is an important task in modern biology since the theory of evolution published by *Charles Darwin* became widely accepted. In the past the relation of two species could only be estimated by their *morphology*, the shape of the bodies of their individuals. Today we can sequence and compare their genomes, allowing for a detailed analysis of the genes they share and where they differ. This allows to reconstruct the relationship of species even if their morphological differences allow different possibilities.

2.4 Homology of Genes

Not only species but also genes evolve. Two genes are called *homologous* if they originate from the same gene, their *most recent* or *lowest common ancestor*. Homologous genes are further separated into two groups. If a gene is duplicated, the original gene and its new copy are called *paralogous*. If two genes originating from the same ancestor are separated by a speciation event, they are called *orthologous*.

Orthologous genes, or *orthologs* for short, usually fulfill a similar function in their respective species. Therefore, they diverge only slowly and their sequences show a high similarity. This fact is used by most orthology detection tools for finding orthologous candidates. It also allows to predict the function of a gene, if the function of an orthologous gene in another species is already known.

Paralogous genes can be separated into two groups: *out-paralogs*, which arose by duplication *before* a certain speciation event, and *in-paralogs*, which arose *after* the speciation. Usually, in- and out-paralogy is defined with respect to the most recent common speciation event.

Genes are called *co-orthologous* with respect to a certain gene, if they are all orthologous to that gene. Sets of co-orthologs arise e. g. if in-paralogs are orthologous to the same gene.

3 Mathematical Preliminaries

This section will give the basic mathematical definitions and results used throughout this thesis. Additionally, it will be noted how these will be used to model biological concepts.

3.1 Undirected and Directed Graphs

An *undirected graph* G is a tuple (V, E) of two sets V and E. The set V is called the *vertex set* whereas E is called the *edge set*. The vertices are also called *nodes*. The elements $e \in E$, the edges, each connect two distinct vertices from the vertex set with each other. If an edge e connects the vertices u and v, then $e = \{u, v\}$. Also $V(G)$ and $E(G)$ denote the vertex set and edge set of G, respectively. Note that $\{u, v\} = \{v, u\}$ is a set and thus has no order and contains each element at most once. Therefore, an edge always connects two *distinct* nodes. Also the edges have no direction.

A *directed* graph, in contrast, has an edge set E consisting of tuples (u, v) for distinct vertices u, v from its vertex set. Therefore, an edge (u, v) is *directed* from u to v while the edge (v, u) is directed from v to u.

Graphs, especially smaller ones, can easily be visualized. When drawing graphs, nodes are usually represented as circles and edges as lines or curves connecting them. Directed edges are marked with an arrow pointing to the vertex the edge is pointing to, see figure 3 for an example of an undirected and a directed graph.

If two nodes of a graph are connected by an edge, they are said to be *adjacent*. An edge e connecting two nodes u, v is *incident* to these two nodes, and vice versa u and v are incident to e.

In biology, graphs are commonly used to model many different kinds of interactions and networks, e. g. gene-regulatory networks, protein-protein interaction or food chains. In bioinformatics, they are an essential tool to algorithmically solve many different biological problems, including (consistency-based) multiple sequence alignments. In this work, graphs will be used to model gene relationships by choosing a set of genes X as vertex set and connecting orthologous genes with an edge. Such a graph is called the *orthology graph* for the genes on X.

3.1.1 Complements, Paths and Connected Components of Graphs

For a given graph G, its *complement* $\overline{G}$ is defined as the graph with $V(\overline{G}) = V(G)$ and $E(\overline{G})$ contains exactly those edges that $E(G)$ does not contain. Figure 4 shows the complement of the graph shown in figure 3a.

A graph G is called a *subgraph* of a graph H, if $V(G) \subseteq V(H)$, $E(G) \subseteq E(H)$ and $\forall \{u, v\} \in E(G) : \ u, v \in V(G)$. This last condition means that G may only contain those edges of H whose incident vertices u, v are in $V(G)$, too. G is called *induced* subgraph of H, if G is a subgraph of H and additionally $\forall u, v \in V(G) : \ \{u, v\} \in E(G) \Leftrightarrow \{u, v\} \in E(H)$ holds, i. e. the nodes of G are connected by an edge in G if

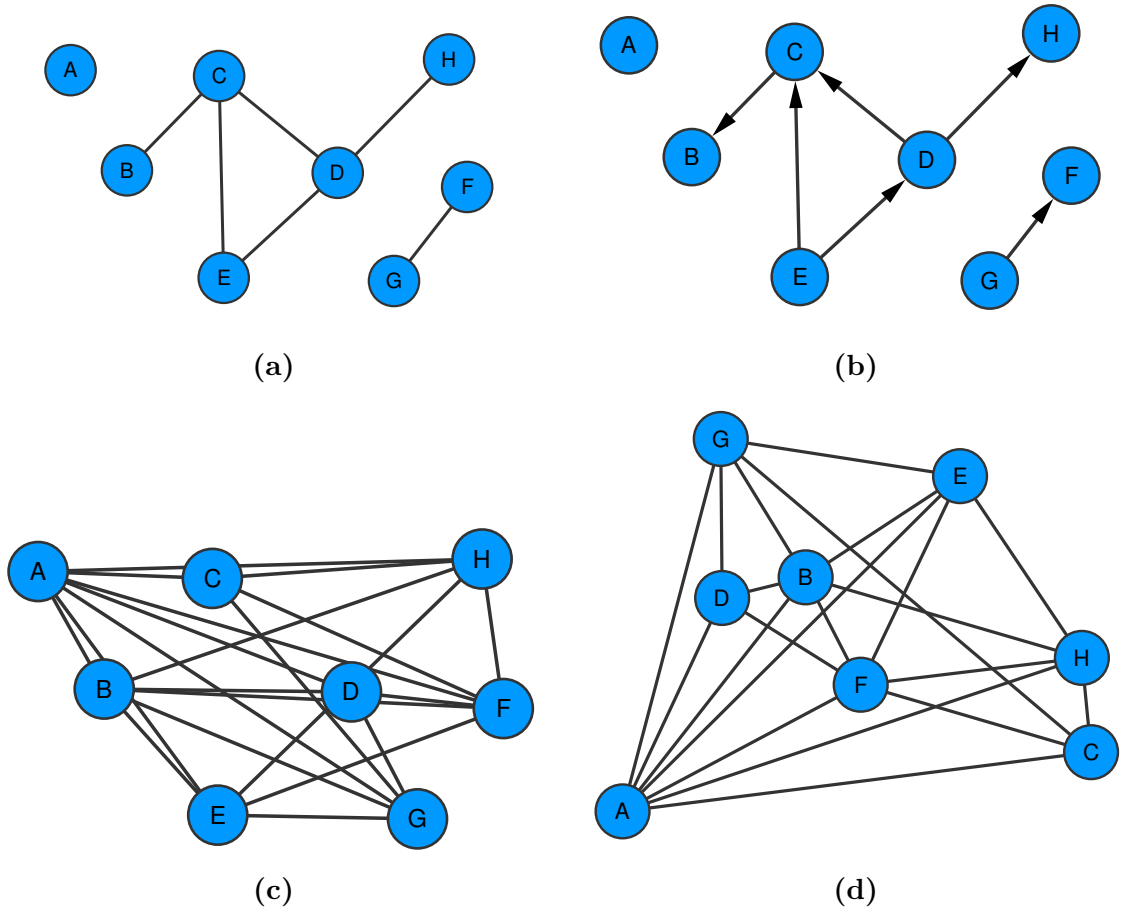


Figure 3: An undirected (a) and a directed (b) graph on the same vertex set $V = \{A, \ldots, H\}$. The arrows indicate the direction of the edge, e. g. edge (C, B) is directed from node C to node B.

Additionally, the complement of graph (a) is shown in (c). Its edge set contains exactly all the edges that are not in the edge set of (a). In (d) the same graph is shown with a different layout to improve the readability. Nevertheless, this example demonstrates that the visualization of bigger graphs can be a difficult problem.

and only if they are connected in H. An induced subgraph on G is fully defined only by its vertex set.

A *path* in $G = (V, E)$ is a sequence $p = v_1 e_1 v_2 \cdots e_{n-1} v_n$ of nodes $v_1, \ldots, v_n \in V$ and edges $e_1, \ldots, e_{n-1} \in E$ such that $\forall i \in \{1, \ldots, n-1\} : \; e_i = \{v_i, v_{i+1}\}$ ($e_i = (v_i, v_{i+1})$ for directed graphs, respectively), i. e. two consecutive nodes in the sequence are incident to the edge between them. p is a path from v_1 to v_n and has the *length* n, which is the number of its vertices.

In a *connected* graph $G = (V, E)$, for any two nodes $u, v \in V$ there exists a path from u to v. A graph that is not connected is *disconnected*. A *connected component* C of G is a *maximally connected*, induced subgraph of G. Maximally connected means that after removing any vertex to C, it is no longer connected. The graph in figure 3a has three connected components: the induced subgraphs defined by the vertex sets $\{A\}$, $\{B, C, D, E, H\}$ and $\{F, G\}$.

Two graphs G and H are called *disjoint* if their vertex sets are disjoint, i. e. if $V(G) \cap V(H) = \emptyset$ holds.

3.1.2 Multipartite graphs

For a set S, a *partition* is a decomposition of S into multiple subsets $S_1, \ldots, S_k$ with $S_1 \cup S_2 \cup \ldots \cup S_k = S$ that are pairwise disjoint ($S_i \cap S_j = \emptyset \; \forall i \neq j$, $i, j \in \{1, \ldots, k\}$).

An undirected (directed) graph $G = (V, E)$ is called *multipartite*, if there is a partition of its vertex set V such that for any edge $e = \{u, v\}$ ($e = (u, v)$) its incident nodes u and v belong to different subsets of the partition. In other words, the nodes are divided into different groups and there are no edges connecting two nodes belonging to the same group. If the partition has exactly two subsets, the graph is called *bipartite*. Figure 4 shows a multipartite graph whose nodes are partitioned into three groups.

The notion of multipartite graphs is used by `Proteinortho`, an orthology detection tool described in section 4.2, to identify clusters of co-orthologous genes in an orthology graph.

3.1.3 Representations of Graphs

Since in Bioinformatics computers are used to process graphs, a representation of graphs that is machine readable is required. There are many of those, but in this work two of them will be used.

An *adjacency list* is a list (i. e. a text file) where each entry (i. e. line) is the name of a vertex, followed by the names of the nodes that are adjacent to it. An adjacency list could look like this:

```
A | B D
B | A
C |
D | A
```

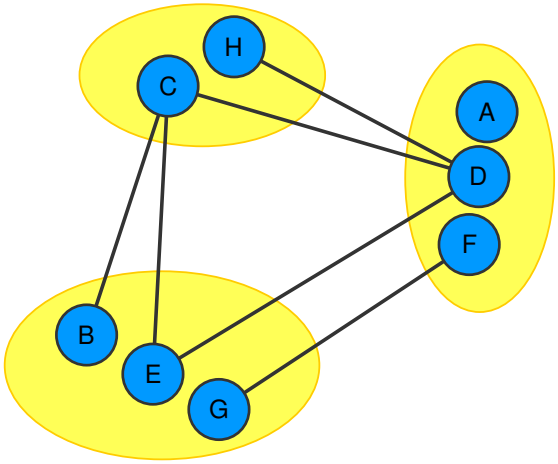


Figure 4: A multipartite graph with a vertex partition consisting of three subsets (marked yellow). It is the same graph shown in figure 3a.

This is the adjacency list of a graph $H = (V', E')$ with $V' = \{A, B, C, D\}$ and edges connecting A and B, and A and D. C is an isolated vertex and the pipe " | " is a separator. For directed graphs, the edges are always directed from the first vertex of each line to the other ones following the pipe. For undirected graphs, it needs to be ensured that if any vertex u is marked as adjacent to v, v needs to be marked as adjacent to u, too. An adjacency list of an arbitrary undirected graph $G = (V, E)$ has the size $|V| + 2|E|$, since the list needs $|V|$ entries and each edge must be added to both of its incident vertices. This allows to access incident nodes of a given vertex in constant time if its position in the list is known. Especially for storing sparse graphs the adjacency list is a good choice.

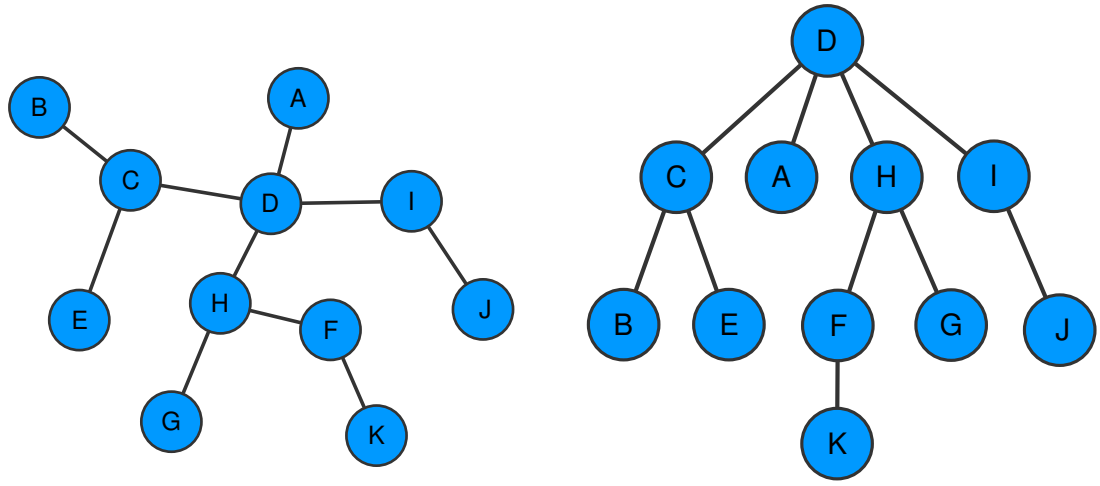
Another possibility to store graphs is the *adjacency matrix*. It is a matrix $A \in \{0, 1\}^{|V| \times |V|}$, i. e. a *Boolean* matrix consisting of $|V|$ rows and columns of 0 and 1 where each $v \in V$ is assigned a (pairwise distinct) position $pos(v)$ from 1 to $|V|$ (pos is bijective). Let (a_{uv}) denote the entry of A in row $pos(u)$ and column $pos(v)$. Then $(a_{uv}) = 1$ if $(u, v) \in E$, else it is 0. The adjacency matrix of H is given by:

```
0 1 0 1
1 0 0 0
0 0 0 0
1 0 0 0
```

The positions of the vertices were chosen according to their lexicographic order. The entries of the main diagonal of any adjacency matrix are always 0 since no vertex is adjacent to itself. The adjacency matrix requires a space of size $|V|^2$, no matter how many edges the graph has. If the positions of two nodes u, v in the graph is known, checking whether they are adjacent can be performed in constant time using an adjacency matrix.

3.2 Trees

Let $T = (V, E)$ be a graph and $v \in V$) be a nodes. A *cycle* is a path of length at least two from v to v, i. e. the path starts and ends at the same vertex. T is said to be *acyclic* if it does not contain any cycle. An acyclic graph that additionally is connected is called a *tree*. For an example of a tree see figure 5a.



(a) **(b)**

Figure 5: An unrooted (a) and a rooted (b) tree on the same vertex set $V = \{A, \ldots, J\}$ with identical edge sets. Unlike the graph shown in figure 3a, they are connected graphs and do not include any cycles. Both graphs are identical (*isomorphic*) in the sense that only their layout differs. In (b), node D has been chosen as root, the nodes $\{B, E, K, G, J\}$ are leaves and the remaining ones are internal nodes. The *Newick* notation of this tree is given by $((B, E)C, A, ((K)F)H, (J)I)D$.

The edges in a tree T can be given a direction. This is done by choosing an arbitrary node $\rho \in V$ that is now called the *root* of T. All edges are pointing away from the root. Therefore, any node v of a rooted tree has an *in-degree* (*out-degree*), defined as the number of edges incident to v that point to v (away from v, respectively). ρ is the only node with an in-degree of 0. The vertices having an out-degree of 0 are called the *leaves* of T. leaves(T) denotes the set of leaves in T. Nodes that are neither leaves nor the root are called *internal nodes*. Note that for unrooted trees, the notion of leaves refers to the vertices incident to exactly one edge, while the other vertices are the internal nodes. For any node v of a rooted tree, the set of nodes that are incident to an outgoing edge v is the set of *successors* or *children* of v. The single adjacent node connected to v by an incoming edge is called the *predecessors* or *father* of v.

Trees can be visualized just like graphs, but in contrast to directed graphs the edge direction of rooted trees is represented implicitly by drawing the root on top

and adding the successors of each node below their predecessor, as shown in figure 5.

Since trees are acyclic and connected, there is exactly one path from one node to another one. For any two nodes u, v in a tree with root ρ, the paths from u to ρ and from v to ρ will partially overlap, and the first common node of both paths is called the *lowest common ancestor* $\mathrm{lca}(u, v)$ of u and v. For example, in the tree in figure 5b, $\mathrm{lca}(K, G) = H$.

Trees are a versatile tool for modeling relations, especially hierarchical ones like file systems or family trees. In biology, they are used to represent phylogenetic trees, where the nodes are species and an edge (A, B) means that species B evolved (directly) from species A. Reconstructing the "tree of life", i. e. the phylogenetic tree containing all recent and extinct species of our planet, is a major task of modern biology.

3.2.1 Gene trees

Just like phylogenetic trees, gene trees model the evolutionary history, but on the level of single genes. Let X be a set of homologous genes where for any two nodes from X it is known whether they are orthologous or paralogous to each other. A *labeled gene tree* T on X is a rooted tree with $\mathrm{leaves}(T) = X$ such that a mapping $f : V(T) \setminus X \to \{0, 1\}$ exists that assigns to each internal node a label, either 1 or 0. This label shall have the property that the lowest common ancestor $\mathrm{lca}(x, y)$ of any two genes $x, y \in X$ will have the label 0, if x, y are paralogs, and 1 if they are orthologs.

Gene trees model the evolutionary history of genes just like phylogenetic trees do for species. An internal node labeled with 1 represents a speciation event, while the label 0 indicates a gene duplication event. The reason why gene trees are interesting is that they can be used to reconstruct the underlying species tree in polynomial time as described in [HR+12]. Though in general this reconstruction is ambiguous, it is nevertheless a useful tool to gain information on the phylogeny of a set of species, if their annotated genomes are available for analysis.

Indeed, the construction of a proper gene tree will be the goal of this work. As it turns out, the information stored in a labeled gene tree of a set of genes X is already contained in the orthology relation of the genes in X [Hel+13], but this will be explained later.

3.2.2 *Newick* Notation

Though trees can be stored in any graph representation described in section 3.1.3, their special structure can be used for a more efficient storage. A very popular form is the *Newick notation*. It is defined recursively as follows for any tree T with root ρ:

- If ρ is a leaf, $\mathrm{Newick}(\rho) = \rho$.
- Else, ρ has child nodes $\rho_1, \ldots, \rho_k$. Then

$$\mathrm{Newick}(\rho) = (\mathrm{Newick}(\rho_1), \ldots, \mathrm{Newick}(\rho_k))\rho.$$

The label ρ after the closing parenthesis can also be omitted, or set to e. g. 0 and 1 to indicate duplication or speciation events in a gene tree. To achieve a unique representation, the nodes $\rho_1, \ldots, \rho_k$ should be ordered, e. g. according to their lexicographic order.

As an example the *Newick* notation of the tree in figure 5b is given there. As an additional example, the representation of the tree in figure 6b is given by

$$\Big(\big(A, B\big)1, \big((C, E, D)0, (F, G)0\big)1\Big)0.$$

This notation is required for the tool `splitDist` (see section 4.5), which is used in this work for comparing reconstructed gene trees with the original ones. Note that there exist variations of this notation since there is no official standard describing it.

3.2.3 Split Distance

Given two trees T_1, T_2 on the same leaf set X, one can ask how different they are. In the context of this work, this question arises in this form: given two gene trees T_1, T_2 on a set of genes X, how big is the distance between the evolutionary scenarios those trees describe? Another use for such a distance measure is the comparison of two phylogenies on the same set of species.

A well-known distance measure for trees is their *split distance*. Let $T = (V, E)$ be a tree and $e = \{u, v\} \in E$ be an edge of T. According to [Mai], the *split* induced by e is the partition of the leaf set X of T into two subsets X_1, X_2 such that X_1 (X_2) contains exactly those leaves that are in the component of $(V, E \setminus e)$ containing u (v, respectively). More intuitively, each edge splits the leaf set into two groups containing the leaves on each side of this edge. For example, the tree shown in figure 5a contains the split $\{A, J, B, E\}, \{G, K\}$, induced by the edge $\{D, H\}$. The splits induced by edges incident to a leaf are called *trivial splits*. Obviously, any two trees on the same leaf set share the same trivial splits.

Let $S(T)$ denote the set of splits in T. Now, the *split distance* of T_1 and T_2 can be defined as the number of splits in T_1 not found in T_2. Since this absolute number strongly depends on the total number of splits in the trees, a normalized variant is more useful for this work. The normalized split distance $\mathrm{sdist}(T_1, T_2) \in [0, 1]$ is defined as the number of splits in T_1 not contained in T_2, divided by the total number of splits in T_1, i. e.

$$\mathrm{sdist}(T_1, T_2) = \frac{|S(T_1) \setminus S(T_2)|}{|S(T_1) \cup S(T_2)|}.$$

There are several tools available to compute the split distance for given trees. In section 4.5, one called `SplitDist` is further described. Though in this work and in the mentioned tool the above definition of the split distance is used, there is an alternative and more common definition based on the *Jaccard* index. It defines $\mathrm{sdist}_{alt}(T_1, T_2)$ as the symmetric difference of the splits in T_1 and T_2, divided by the

total number of splits in T_1 and T_2, so

$$\text{sdist}_{alt}(T_1, T_2) = \frac{|S(T_1) \triangle S(T_2)|}{|S(T_1) \cup S(T_2)} = \frac{\left|\left(S(T_1) \setminus S(T_2)\right) \cup \left(S(T_2) \setminus S(T_1)\right)\right|}{|S(T_1) \cup S(T_2)|}.$$

Note that the definition used here is, in contrast to the alternative one, not symmetric, so in general $\text{sdist}(T_1, T_2) \neq \text{sdist}(T_2, T_1)$.

The split distance is one of the two measures used to evaluate the results of the tool developed in this work. It was chosen because it is popular and tools for its calculation are available. However, it might not be optimally suited for the task of comparing gene trees. The major concern is that the split distance neglects the labels 0 and 1 marking orthology and paralogy in the tree. Two gene trees can have the same edge set, but the labels might be swapped, i. e. 0 instead of 1 and vice versa.

3.3 Cographs

Cographs are a well-known graph class that has been studied since the early 1970s [CLB81]. They were studied independently in many different areas of mathematics and therefore have various names or equivalent characterizations. Alternative names include D^*-graphs or *Hereditary Dacey* graphs.

A *cograph* is a graph G that is defined recursively as follows [Liu+11]:

- the graph consisting of only one node (the *trivial graph*) is a cograph
- for any cograph G, $\overline{G}$ is also a cograph
- for any any two disjoint cographs G, H, $G \cup H$ is also a cograph

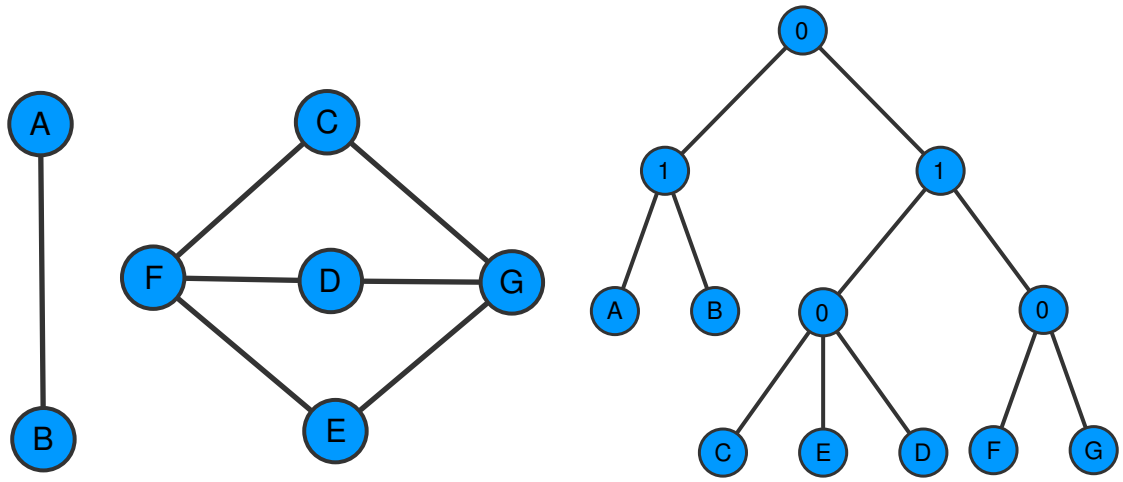
The union $G \cup H$ of two graphs G, H is the graph with the vertex set $V(G) \cup V(H)$ and the edge set $E(G) \cup E(H)$. For disjoint G and H, $V(G) \cap V(H) = \emptyset$ holds. Figure 6a shows an example of a cograph.

3.3.1 Cographs and the Orthology Relation

Cographs are closely related to gene orthology. Let X be a set of homologous genes. Then a mapping $\delta : X \times X \to \{0, 1\}$ can be constructed such that for distinct genes $x, y \in X$, $\delta(x, y) = 1$ if x, y are orthologs, and $\delta(x, y) = 0$ if they are paralogs. As recently shown by [Hel+13], δ is a so called *symbolic ultrametric*. Furthermore, they proved that for any symbolic ultrametric on X, the graph $G = (X, E)$ with

$$E = \{\{x, y\} \mid x, y \in X,\, \delta(x, y) = 1\}$$

is a cograph. This means that any orthology graph is a cograph. Therefore, the mathematical properties of cographs are particularly interesting for the reconstruction of the orthology relation of X.



(a) **(b)**

Figure 6: A cograph (a) and its associated cotree (b). The cograph does not contain an induced P_4. The cotree has strictly alternating labels and can be used to reconstruct the cograph (a).

3.3.2 Equivalence to P_4-free Graphs

Two graphs $G = (V, E), G' = (V', E')$ are called *isomorphic* to each other, if there exists a bijective mapping $\mathrm{f} : V \to V'$ such that $\forall u, v \in V : \{u, v\} \in E \Leftrightarrow \{\mathrm{f}(u), \mathrm{f}(v)\} \in E'$. Intuitively, this means that both graphs can be drawn the same way, except for the names of their nodes.

A P_4 is a graph (V, E) with $V = \{u, v, w, x\}$ and $E = \Big\{\{u, v\}, \{v, w\}, \{w, x\}\Big\}$, or any other graph isomorphic to it. In other words, P_4 is a path consisting of four nodes and three edges.

It can be shown that the graphs not containing any P_4s as induced subgraph are exactly the cographs [CLB81]. This provides an intuitive characterization of them, which will be useful later on. Of course, the cograph in figure 6a does not contain any P_4s.

3.3.3 Modular Decomposition

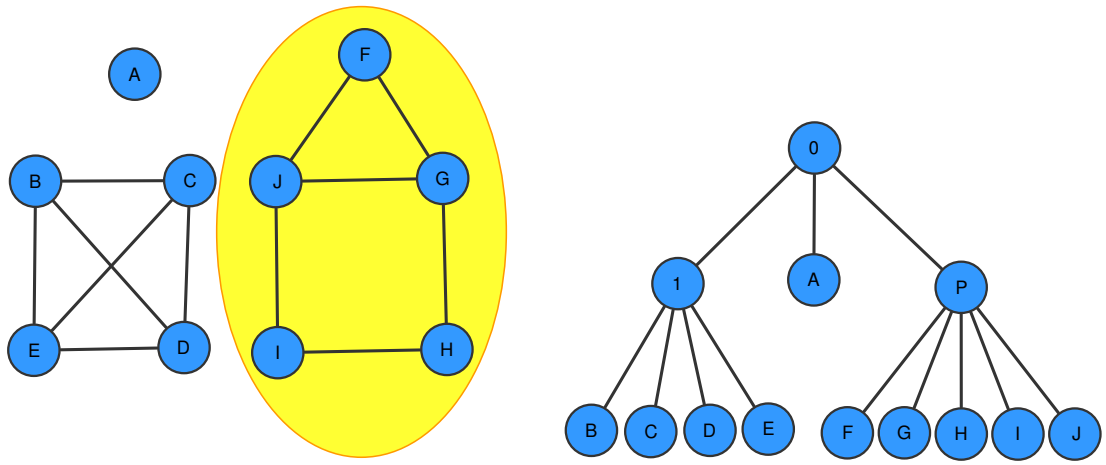
To further analyze the properties of cographs, a certain decomposition procedure for graphs called the *modular decomposition* is required. An arbitrary *non-trivial* graph G (i. e. G has more than one vertex) can be decomposed into so-called *modules*. If G is connected and its complement $\overline{G}$ is disconnected, G is called a *series module*. If G is disconnected and its complement $\overline{G}$ is connected, G is called a *parallel module*. Otherwise, G is a *prime module*.

Let $G_1, \ldots, G_n$ denote the connected components of G. Then either $n = 1$ (i. e. $G = G_1$ is connected) or $n > 1$ (i. e. G is disconnected). Also, let $G'_1, \ldots, G'_k$ be the connected components of $\overline{G}$. Using the above definitions, the *canonical modular*

decomposition tree $T(G)$ can be defined in *Newick* notation:

$$T(G) = \begin{cases} v & \text{if } G \text{ is trivial, } \{v\} = V(G) \\ (T(G_1), \ldots, T(G_n))0 & \text{if } G \text{ is a parallel module} \\ (T(\overline{G'_1}), \ldots, T(\overline{G'_n}))1 & \text{if } G \text{ is a series module} \\ (v_1, \ldots, v_n)P & \text{if } G \text{ is prime, } \{v_1, \ldots, v_n\} = V(G) \end{cases}$$

The leaves of the decomposition tree are exactly the single vertices. The 0 and 1 behind the parenthesis in the parallel and series module cases are labels for the internal node of the tree. The label P marks prime modules. They appear only as the last internal node on any path from the root to a leaf. For on example of a graph and its associated decomposition tree, see figure 7.



(a) **(b)**

Figure 7: (a) This graph is a parallel module, consisting of the trivial graph $(\{A\}, \emptyset)$, the complete graph on the vertices B, C, D and E, and a prime module which has been highlighted. (b) This is the decomposition tree of (a), were the children of node 1 form a series module, the children of 0 form a parallel module and the children of P form a prime module.

In the canonical tree, the labels 1 and 0 alternate on any path from the root to a leaf, i. e. if an internal node has label 1, any of its internal successor nodes is labeled 0 and vice versa, unless it is a prime or a trivial module. To verify that, assume that G is a parallel module with components $G_1, \ldots, G_n$, $n > 1$, so G gets the label 0 in the modular decomposition tree. If there was a component G_i, $1 \leq i \leq n$ with label 0, it would be a parallel module and thus disconnected, contradiction. Assume now that G is a series module, $G'_1, \ldots, G'_n$ are the components of its complement and $G^* = \overline{G'_i}$, $1 \leq i \leq n$ is a series module, too. Then G^* is connected and $\overline{G^*}$ is disconnected, but $\overline{G^*} = G'_i$, and G'_i is connected by assumption, contradiction. Therefore, the claim holds. □

3.3.4 Cotrees

The definition of the modular decomposition tree looks somewhat similar to the definition of cographs. So given a cograph G, what will $T(G)$ look like? First, consider the following lemma:

Lemma. *A non-trivial cograph G is connected if and only if its complement is disconnected.*

The lemma trivially holds if $|V(G)| = 2$, since $\overline{(\{u, v\}, \emptyset)} = (\{u, v\}, \{\{u, v\}\})$ and those are the only two graphs with two nodes, up to isomorphism. If $G = G_1 \cup G_2$ for two disjoint cographs G_1, G_2, then G is disconnected. Therefore, any two vertices $u \in G_1$ and $v \in G_2$ are non-adjacent in G and adjacent in $\overline{G}$. Also, any two vertices $s, t \in G_1$ ($s, t \in G_2$) are indirectly connected in $\overline{G}$ since both s and t are adjacent to an arbitrary node $w \in G_2$ ($w \in G_1$, respectively) in the complemented graph. It follows that $\overline{G}$ is connected. Note that this argumentation is valid for any disconnected graph with at least two nodes, not just for cographs.

The other case is that $G = \overline{G'}$ for a cograph G' that by induction satisfies the claim. Then $\overline{G} = G'$ is disconnected if and only if $G = \overline{G'}$ is connected. □

The consequence of this lemma is that any cograph is either a single vertex, a series module or a parallel module, but not a prime module. Since any induced subgraph of a cograph is a cograph, too, this means that a cograph cannot contain a prime module. Because of that, the decomposition tree of a cograph, also called its *cotree*, has strictly alternating labels of 1 and 0 at its internal nodes and single vertices as leaves. Figure 6b gives an example of a cotree.

Consider also the connection of prime modules and induced P_4s. Any prime module must contain at least one P_4, otherwise it would be a cograph and no prime module. On the other hand, any P_4 must be contained in a prime module, since otherwise a cograph that contains a P_4 could be constructed. Therefore, modular decomposition can be applied to an arbitrary graph to find its prime modules and thus (approximately) locate the P_4s, which is just the approach that is used in this work to tackle the problem defined in the next sections.

3.3.5 Reconstructing Cographs from their Cotrees

By the section above, it follows that a cograph is uniquely represented by and can be reconstructed from its associated cotree. To do that, one needs to start at the leaves of the cotree and interpret them as trivial graphs containing this single node. Then each internal node G with the children $G_1, \ldots, G_n$ represents the graph $G_1 \cup \ldots \cup G_n$ if G is labeled with 0, or $\overline{\overline{G_1} \cup \ldots \cup \overline{G_n}}$ if G is labeled with 1. The latter operation is called the *join* of $G_1, \ldots, G_n$ and can also be thought of as taking $G_1 \cup \ldots \cup G_n$ and adding all edges that connect any two vertices $u \in G_i$ and $v \in G_j$, where $i \neq j$.

3.3.6 Cotrees are Gene Trees

There is another useful property of cotrees, as the following lemma shows:

Lemma. *For a cograph G and its cotree $T(G)$ with the associated labeling function $f_{T(G)} : V(G) \setminus \text{leaves}(G) \to \{0, 1\}$,*

$$\{u, v\} \in E(G) \Leftrightarrow f_{T(G)}(\text{lca}_{T(G)}(\{u, v\})) = 1$$

holds for all distinct vertices $u, v \in V(G)$.

To verify this, let $u, v \in V(G)$ and $f_{T(G)}(lca(\{u, v\})) = 0$, i. e. the lowest common ancestor of u and v in the cotree of G is labeled wit 0. Then $\text{lca}(u, v)$ is a parallel module and thus u and v are in distinct components of G, so $\{u, v\} \notin E(G)$. Assume now that $f_{T(G)}(lca(\{u, v\})) = 1$, so $\text{lca}(u, v)$ is a series module and any two nodes from distinct children of $\text{lca}(u, v)$ are connected due to the properties of the join operation. u and v cannot belong to the same child of $\text{lca}(u, v)$, because then the latter would not be the lowest common ancestor of u and v in $T(G)$. □

In a biological context, i. e. when G is a valid orthology graph, this means that in the cotree $f_{T(G)}(lca(\{u, v\})) = 1$ if and only if the genes u, v are orthologous. This means that the cotree $T(G)$ is a gene tree of the genes in $V(G)$.

3.3.7 Cograph Editing

The *cograph editing problem* is defined as follows:

Instance: An undirected graph $G = (V, E)$.

Question: Find edge sets $E^+ \subseteq \binom{V}{2} \setminus E$ and $E^- \subseteq E$ such that $\widehat{G} = (V, E \cup E^+ \setminus E^-)$ is a cograph. $\binom{V}{2}$ is the set of all edges connecting two distinct nodes of V.

Cost: Minimize $|E^+| + |E^-|$.

In other words, edges are added to (E^+) or removed from (E^-) G to transform it into a cograph. The edges in E^+ and E^- are called *edition operations* and are *applied to* G, i. e. they are either added or removed. Together, the edges from both sets form an *edition set* for G.

As an example, consider the graph from figure 8a. Obviously, it is not a cograph since e. g. the nodes B, C, D and H form an induced P_4. There are numerous cograph edition sets for this graph, one example would be $E^+ = \{\{H, G\}\}$ and $E^- = \{\{C, D\}, \{C, E\}, \{D, E\}\}$. The minimal edition set is given by $E^+ = \emptyset$ and $E^- = \{\{D, H\}\}$.

Obviously, such edition sets exist for any graph G, since adding all missing edges to it will yield a complete graph, which is a cograph since it does not contain an induced P_4. The other extreme case, removing all edges from G, will also result in a cograph for the same reason. Instead of such trivial solutions, one is usually interested in edition sets that meet certain criteria. Finding a *minimal edition set* is the most common task, i. e. finding a set containing as few edges as possible. Unfortunately, it has been shown that finding an edition set of at most k edges is NP-complete [Liu+11], so it is unlikely that there exists a polynomial-time bounded

algorithm to solve this task. Note that k is treated as input here. However, this problem is *fixed-parameter tractable*, which means that if k is a fixed integer, the problem can be solved in polynomial time. The complexity increases exponentially only if k is increased. An implementation of such an algorithm is `coedit`, described in section 4.1

3.3.8 Cograph Editing of Orthology Graphs

As described in section 3.2.1, gene trees can be used to reconstruct phylogenetic trees. Further, is has been explained that an orthology graph is a cograph (section 3.3.1), and that a gene tree can be reconstructed from an orthology graph easily by modular decomposition (see section 3.3.3).

Piecing together the puzzle, all that is left to do to reconstruct a possible phylogeny for a number of species from the sequences of their genes or proteins is building their orthology graph. Since the true orthology relation is unknown, it has to be estimated. There are a number of tools for this task, but here a program called `POFF` will be used, see section 4.3 for an exact description of how it works. In a nutshell, it estimates which genes could be orthologous based on their sequence similarity and their position on the chromosome. However, the result of all such tool is always an approximation and there will be genes marked as orthologous which are not, as well as genes which are in fact orthologs but are not reported as such. Because of this inevitable errors, the graph G constructed from the estimated orthology relation will most likely not be a cograph.

Since a cograph is required for further processing of the input data as suggested above, the only solution is to modify G in such a way that it becomes a cograph. This could be done by computing a minimal edition set for G, but since in practical applications G can have tens of thousands of vertices, with an exponential-time algorithm there is no hope of finding an optimal solution within a feasible amount of time. The focus of this work therefore lies on developing a heuristic algorithm that can deal with such huge graphs.

3.3.9 Cograph Completion

Another problem related to cograph editing is *cograph completion*. Again, an arbitrary graph $G = (V, E)$ is supposed to be transformed into a cograph $\widehat{G}^+$ but, as the name suggests, only by *adding* edges to Gs edge set:

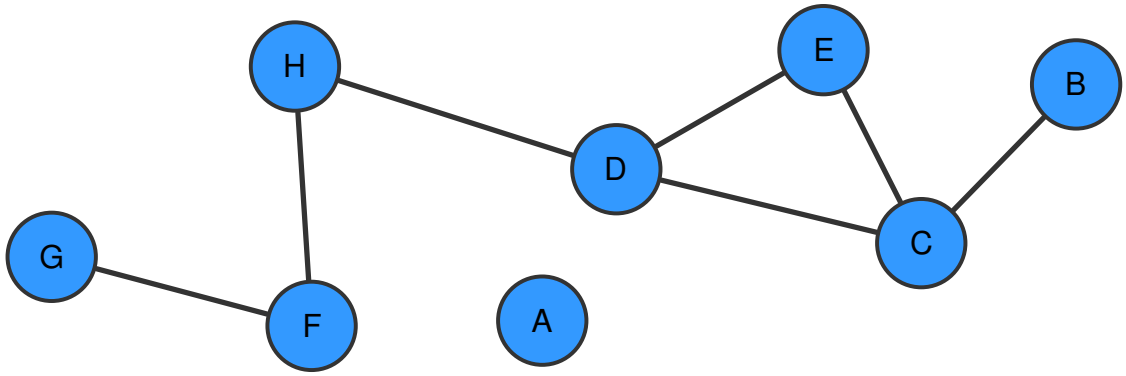
Instance: An undirected graph $G = (V, E)$.

Question: Find an edge sets $E^+ \subseteq \binom{V}{2} \setminus E$ such that $\widehat{G} = (V, E \cup E^+)$ is a cograph.

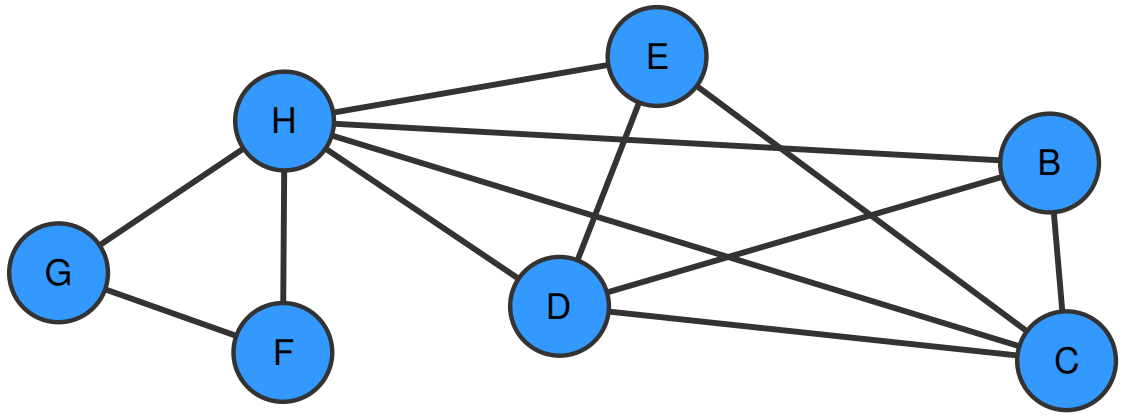
Cost: Minimize $|E^+|$.

For example, to transform the graph from figure 8a into a cograph, the cograph completion set $E^+ = \{\{B, D\}, \{B, H\}, \{C, H\}, \{E, H\}, \{G, F\}\}$ is applied to it. The result is shown in figure 8b.

For this problem it is unknown whether there is a polynomial-time algorithm to find a minimal edition set, however, there is an algorithm that can compute an *inclusion minimal* solution in linear time with respect to $|V| + |E|$ [LMP08]. An edition set E^+ is inclusion minimal if no proper subset of E^+ exists that is by itself an edition set for G. Especially, this means that the edition set resulting from the application of this algorithm may not have the minimal number of edges with respect to all other edition sets, and thus, is not minimal in the sense used here to describe cograph edition sets. Nevertheless, the authors suggest that, because of the fast running time and its probabilistic character, the algorithm can be applied multiple times and from these samples the best solution can be chosen.



(a)



(b)

Figure 8: (a) A graph containing multiple induced P_4s, i. e. it is not a cograph. A cograph edition set is given by $E^+ = \{\{H, G\}\}$ and $E^- = \{\{C, D\}, \{C, E\}, \{D, E\}\}$, the minimal edition set is given by $E^+ = \emptyset$ and $E^- = \{\{D, H\}\}$. (b) Here, the cograph completion set (see section 3.3.9) $E^+ = \{\{B, D\}, \{B, H\}, \{C, H\}, \{E, H\}, \{G, F\}\}$ has been applied to the graph from figure (a).

4 Third Party Tools and Algorithms

The tools of other authors have been used as a foundation for this work. In this section it will be explained how they work.

4.1 `coedit`

As described in section 3.3.7, cograph editing is an NP-complete but fixed-parameter tractable problem for the parameter k, where k is the maximal number of edges in an edition set. An algorithm with a time complexity of $\mathcal{O}(4.612^k + |V|^{4.5})$ has also been proposed by [Liu+11].

The approach is to first edit the input graph to become a P_4-*sparse graph*, a graph where each induced subgraph of exactly five nodes contains at most one induced P_4. To a P_4-sparse graph, cograph editing can be applied in a time of just $\mathcal{O}(|V| + |E|)$. The transformation of an arbitrary graph into a P_4-sparse, however, is complex. It is performed by identifying a number of forbidden subgraphs which are then eliminated by using a search tree approach. For each forbidden subgraph certain edition rules are available, and each application of one of those will mark one edge or non-edge as permanent so it cannot be modified by later edition steps. The algorithm branches with each application such that all possible solutions with a size of at most k will be found. As an additional preprocessing step, before the P_4-sparse editing is applied, a simplification of the input graph called *kernelization* is performed which substitutes parts of the graph by single nodes without changing the outcome of the edition step itself. Figure 9 gives a schematic overview of the entire work flow of the algorithm.

The program `coedit` [Ber12] is a *C++* implementation of the algorithm described above. It will be integrated into the tool developed in this work to deal with small prime modules. Due to the enormous demand for computational power, a small value for k has been chosen (default: $k = 5$). This means that for prime modules of a size of roughly 15 vertices or more no edition set can be found and another method needs to be used.

4.2 `Proteinortho`

`Proteinortho` [Lec+11] is a detection tool for orthologous genes and proteins. Competing applications available before its release had the disadvantage that their memory consumption increased quadratically with the input size. This made the computation for larger datasets infeasible if no high-end hardware was available. `Proteinortho`, however, was designed to run also on modest hardware and made orthology computation available to a wider audience, which became important especially because the advancing sequencing technology produced more and more data. It also supports multiple CPUs and cores.

In this work, `Proteinortho` is not used directly, but `POFF` (section 4.3), which is closely related to it. Therefore `Proteinortho` is explained first.

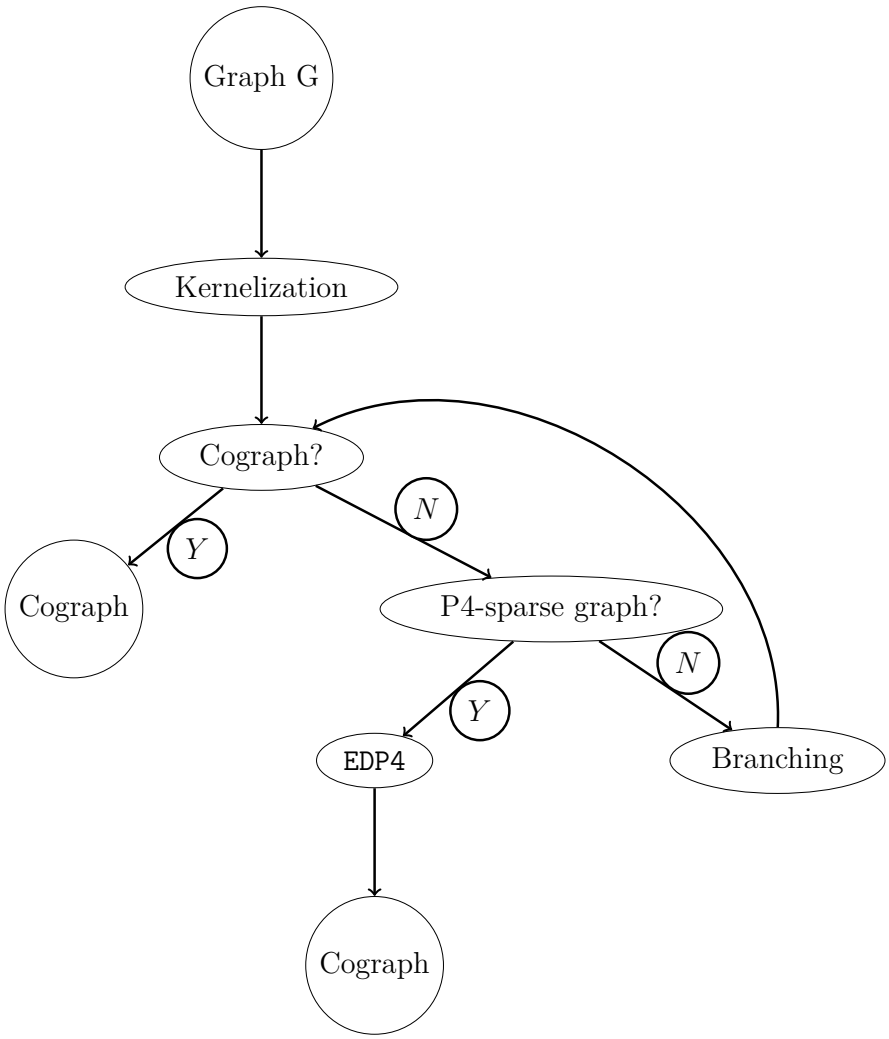


Figure 9: A schematic overview [Ber12] of the cograph editing algorithm of [Liu+11] that is used by `coedit`. `EDP4` is the name of the algorithm that computes a cograph from a P_4-sparse graph.

4.2.1 Constructing a Graph

Given a set of genes, each belonging to a certain species, the sets of co-orthologous genes shall be computed.

At first, a pairwise comparison of any gene with all genes of all other species is performed. This can be done with a tool like `blast`, which produces a *bitscore* for each comparison to measure the sequence similarity of both genes. Those bitscores are then used to calculate the so-called *E-values*, which are generated using a statistical model to represent the probability that two sequences are as similar as they are by chance. This implies that similar sequences produce a small E-value and that short similar sequences receive smaller scores than longer ones. Though they are are not the same, E-values below 0.01 have only a small difference to their statistical *p-value*,

the probability that an E-value occurs by chance.

Once the similarity scores are calculated, groups of orthologous genes are computed based on them. This is done by constructing a weighted, directed graph $\vec{\Upsilon}$ with the genes as nodes. For two nodes x, y an edge is added, if $w((x, y))$, defined as the bitscore `blast` generated when comparing x with y, is above a certain E-value threshold. Finding the orthologous genes corresponds to the task of finding nearly disjoint, maximal, nearly-complete *multipartite* (see section 3.1.2) subgraphs in $\vec{\Upsilon}$.

4.2.2 A Relaxed Reciprocal Best Alignment Heuristic

To extract pairs of ortholog genes from $\vec{\Upsilon}$, a *reciprocal best alignment heuristic* (RBAH) could be used. This means that two genes x and y from species A and B, respectively, are recognized as orthologs, if and only if the score $w(x, y)$ is maximal with respect to any other score $w(x, y')$, $\forall y' \in B$, and the score $w(y, x)$ is maximal with respect to any other score $w(y, x')$, $\forall x' \in A$. This way a symmetric graph Υ_{RBAH} is constructed, containing the genes as nodes and edges connecting exactly those genes which are recognized as orthologous by the RBAH.

The RBAH, however, works correctly only under some assumptions which are not always fullfilled for real data. For example, for a given gene it can find at most one ortholog per species, though several co-orthologs could exist. Furthermore, if two groups of co-orthologous genes exist in two different species, it is possible that, due to small differences in their score, edges between them could be removed in the symmetric graph Υ, see figure 10 for an example.

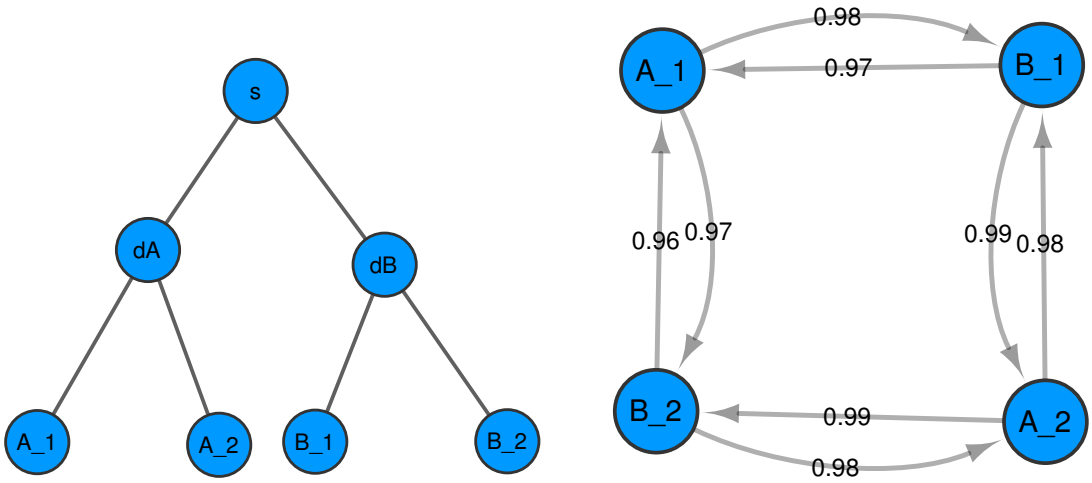
Therefore the RBAH has to be relaxed. A graph $\vec{\Upsilon}^*$ is constructed with a vertex set as before, but a modified edge set such that, for a gene $x \in A$ and another species B, it includes edges to all genes in B that are *almost* as similar as the maximal similar gene from B, so

$$(x, y) \in E(\vec{\Upsilon}^*) \quad \Leftrightarrow \quad \mathrm{w}((x, y)) \geq f \cdot \max_{y' \in B} \Big(\mathrm{w}((x, y'))\Big), \; f \in (0, 1).$$

The default value for the relaxation parameter f is 0.95. This change causes co-orthologs to be included, but at the same time prevents that edges connecting genes showing too much divergence are added to $\vec{\Upsilon}^*$. The undirected graph Υ^* is then extracted from $\vec{\Upsilon}^*$ by retaining an edge between any vertices x and y, if and only if both (x, y) and (y, x) are edges in $\vec{\Upsilon}^*$.

4.2.3 Extracting Groups of Co-orthologs

Now groups of co-orthologs can be found as nearly complete multipartite subgraphs of $\Upsilon*$. Finding complete multipartite subgraphs is in general an NP-complete problem [Cor06], and since in Υ there could be edges missing between genes that are in fact orthologous (*false negatives*), or edges could connect genes that are not (*false positives*), `Proteinortho` solves the task approximately: complete multipartite graphs are dense clusters in the elsewise sparse graph Υ^*. Therefore, decomposing it into its components isolates those subgraphs. If they are sufficiently dense, they are



(a) A gene tree. The root s represents a speciation event, the nodes dA and dB are duplications in species A and B and the leaves are genes in their respective species.

(b) The normalized similarity scores for the genes.

Figure 10: In this example, RBAH fails to add any edges to the symmetric graph Υ^*, though the differences in their similarity scores are negligible.

reported as a group of co-orthologs. If not, then multiple multipartite-like subgraphs are connected by one or a few spurious edges. Those are eliminated using *spectral partitioning*, namely by approximating a minimal cut set using the *Fiedler* vector approach (see e. g. [GM95]).

4.3 `POFF`

The tool `POFF` [LHR+13] is closely related to `Proteinortho`. Actually, newer versions of Proteinortho incorporate the functionality that `POFF`, which is why the current version of `Proteinortho` was used for evaluation in section 6. `POFF` is an orthology detection tool, too, but extends `Proteinortho` by adding *synteny* information evaluation to the detection algorithm, i. e. it uses the relative gene order on the chromosomes as additional filter to reduce the amount of false positive detections. Genetic neighborhood is often conserved during speciation events and can be used to identify duplications that happened before the speciation. This way, large sets of genes that are detected as co-orthologous by `Proteinortho` can be separated into smaller groups of true orthologs by `POFF`. Note, however, that this approach does not work for in-paralogs, where the duplication happened *after* the speciation.

4.3.1 Extracting Synteny Information

There are different algorithms to calculate synteny. Some require gene family information while others rely on pairwise sequence similarity of genes. The heuristic `FFAdj-MCS` [DTS12], which is used by `POFF`, belongs to the latter ones.

`FFAdj-MCS` uses a simple model to represent genomes. Let $\mathcal{G}$ be the set of genes. A Genome G is a sequence of genes $g_i \in \mathcal{G}$, where $1 \leq n \leq n-1$. $|G| = n$ is called the size of the genome. The beginning and end of a genome is marked with the symbol $\circ$ that represents the telomeres, the end of a chromosome. The genes also have a *sign* $(+, -, +$ can be omitted) indicating their direction on the chromosome. Note that this model is *unichromosomal*, but multiple chromosomes can be treated like one large chromosome easily. Assuming that a similarity measure $w : \mathcal{G} \times \mathcal{G} \to [0, 1]$ for any two genes is given, for two given genomes G_1, G_2 a bipartite graph $B = B(G_1, G_2, E)$ can be constructed by taking their genes as vertex set and add an edge e_ij between the genes $g_i \in G_1$ and $g_j \in G_2$ if and only if $w(g_i, g_j) > 0$.

The approach to extract synteny information from B is to calculate a so-called *matching* $\mathcal{M}$, i. e. a set of disjoint edges (no two edges are adjacent to the same vertex), in the bipartite graph $B(G_1, G_2, E)$. Two genes g_i, g_j from a single genome are *consecutive* with respect to $\mathcal{M}$, if they are both incident to (different) edges of $\mathcal{M}$ and no other gene lies between them that is incident to an edge from $\mathcal{M}$. Two pairs of consecutive genes $g_i, g_j \in G_1$ and $g_k, g_l \in G_2$ form a *conserved adjacency*, if $e_{ik}, e_{jl} \in E(\mathcal{M})$ and either

- $k < l$, $\text{sign}(g_i) = \text{sign}(g_k)$ and $\text{sign}(g_j) = \text{sign}(g_l)$, or
- $k > l$, $\text{sign}(g_i) \neq \text{sign}(g_k)$ and $\text{sign}(g_j) \neq \text{sign}(g_l)$,

where $\text{sign}(g)$ is the sign of g. Intuitively, this means that orthologs of g_i, g_j appear in the other genome consecutively (or reversed with a different sign), and therefore their adjacency has been conserved. The goal of determining adjacencies for two genomes can now be achieved by calculating a matching that maximizes both the number of edges and the number of conserved adjacencies. An exact solution can be attained with integer programming, but this problem is NP-hard. Therefore an heuristic approach based on the notion of *maximum common substrings* (MCS) is used.

`FFAdj-MCS` is an iterative algorithm that starts with an empty matching and terminates when there are no more edges between unmatched genes. By default, `POFF` uses only one iteration. In a preprocessing step, all independent genes, i. e. genes that are not adjacent to any other gene, are removed from their respective genome. Then, in each iteration, a common substring of length at least β (default value: $\beta = 3$) of both genomes, up to reversal and switched signs, is calculated that maximizes both the sum of edge weights and the sum of edge weights of conserved adjacencies. The parameter $\alpha \in [0, 1]$ (default value: $\alpha = 0.5$) of `POFF` is used to balance the relative importance of the former two values, though evaluation has shown that its influence is minor. Both sums are added and form the objective

function and `FFAdj-MCS` greedily chooses in each iteration the common substring that maximizes it, the *maximum* common substring. Then, genes in the rest of the genome are deleted, if all their adjacent genes in B are already matched. This heuristic is optimized by extending the MCS of the current iteration on both ends after the deletion step, if possible.

4.3.2 Incorporate Synteny Information into Orthology Detection

Once the matching for two genomes has been computed, it is integrated into the graph construction algorithm of `Proteinortho`. The graph Υ^* is constructed as before (see section 4.2.2), but now, among multiple paralogs which are co-orthologous to a gene x, the matching calculated by `FFAdj-MCS` has selected the one which conserves the local gene order best. `POFF` can therefore split the group of co-orthologous genes into smaller groups of true orthologs.

4.4 Min-Cut Algorithm

Let $G = (V, E)$ be a weighted, undirected and connected graph. A *cut set* $C \subseteq E$ of G is a set of edges such that $(V, E \setminus C)$ is disconnected, i. e. removing the edges of C from G decomposes it into several non-empty components. A cut set is called *minimal* if for any other cut set C' of the graph, $\mathrm{w}(C) := \sum_{e \in C} \mathrm{w}(e) \leq \sum_{e \in C'} \mathrm{w}(e) := \mathrm{w}(C')$ holds. Figure 11 gives an example of a weighted graph for which $\{\{B, D\}, \{C, E\}\}$ represents a minimal cut set. A minimal cut set C decomposes a graph into exactly two components since one edge can connect at most two components, and thus, if applying C resulted in more components, one edge could be removed from C yielding a cut set with less weight, which contradicts the minimality of C. Given two vertices s and t of G, an s-t-cut is a cut-set such that, when it is applied to the graph, s and t lie in different components.

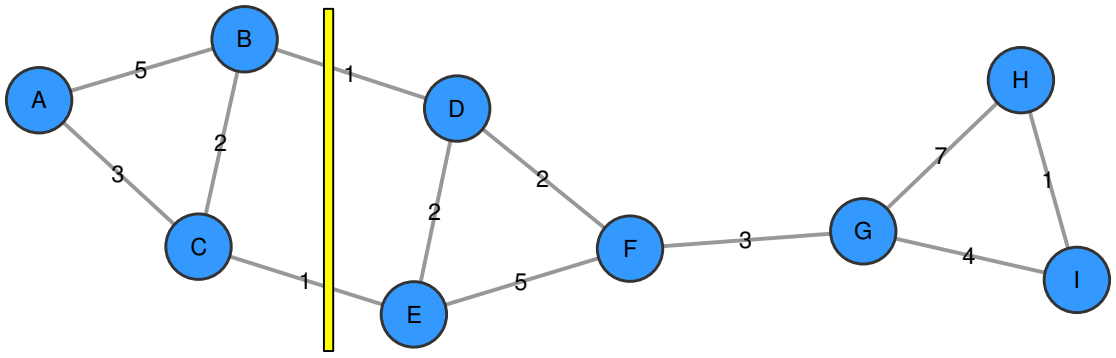


Figure 11: A weighted graph for which the edges $\{B, D\}$ and $\{C, E\}$, marked with a yellow line, form a cut set.

Below a simple algorithm to compute a minimal cut set, or *min-cut*, is introduced [SW97].

4.4.1 Description

The min-cut algorithm presented here is very simple. Its pseudo code is shown in listing 1. `MinimumCut` is called with a connected, undirected, weighted input graph $G = (V, E)$ and outputs a minimal cut-set C for it.

Algorithm 1: MinimumCut(G)

Input: Connected, undirected, weighted input graph $G = (V, E)$
Output: Min-cut of G
$C_{\min} \leftarrow \emptyset$, where $\mathrm{w}(\emptyset) = \infty$;
while $|V| > 1$ **do**
    $C \leftarrow$ `MinimumCutPhase(G)`;
    if $\mathrm{w}(C_{\min}) > \mathrm{w}(C)$ **then**
        $C_{\min} \leftarrow C$;
return $C_{\min}$;

The algorithm iterates while G contains more than one node. In each iteration, the function MinimumCutPhase is called to compute a minimal s-t-cut, and the one with the lowest weight is returned as min-cut.

The function MinimumCutPhase initializes a list A with an arbitrary node a and sucessively appends one node to it in each iteration. More precisely, in every iteration it adds the most tightly connected node of the current A, i. e. it adds a node $v \in V \setminus A$ such that

$$\mathrm{w}(A, v) := \sum_{\{v,u\} \in E, u \in A} \mathrm{w}(\{v, u\})$$

is maximal. In other words, $\mathrm{w}(A, v)$ is the sum of the weights of all edges connecting the node v with any node from the list A, and the most tightly connected node is the node that is currently not in A and maximizes $\mathrm{w}(A, v)$.

Let s and t be the last two nodes added to A. Then all edges incident to t are returned as minimal s-t-cut, and s and t are merged, i. e. they are replaced by a new node st having edges to the same nodes as s and t, with the sum of the weight of both old edges. An edge connecting s and t is removed, if it exists.

4.4.2 Correctness

To show that `minimumCut` indeed calculates a min-cut of G, its authors prove the following lemma:

Lemma. *"Each cut-of-the-phase is a minimum s-t-cut in the current graph, where s and t are the two vertices added last in the phase." [SW97, p. 587]*

Proof. The algorithm generates a linear ordering A of the nodes in G. A node v is called *active* with respect to a cut C if it is not the first node of A and if v and its

Function MinimumCutPhase(G)

Input: Connected, undirected, weighted input graph $G = (V, E)$
Output: Cut-of-the-phase (minimal s-t-cut of G)
$A \leftarrow (a)$, where $a \in V$ is arbitrary;
while $A \neq V$ **do**
  $A.append(v)$, where $v \in V$ is As most tightly connected vertex;
$C \leftarrow \{e | e$ is incident to the node last added to $a\}$;
merge the last two nodes of A in G;
return C;

predecessor in A belong to different components in the graph induced by C. For an arbitrary s-t-cut C (as defined in the lemma) it will be shown that the weight of the cut-of-the-phase is smaller than $\mathrm{w}(C)$. The following notations are used as in [SW97]: "Let $\mathrm{w}(C)$ be the weight of C, A_v the set of all vertices added before v (excluding v), C_v the cut of $A_v \cup \{v\}$ induced by C, and $\mathrm{w}(C_v)$ the weight of the induced cut."

It will be shown below that

$$\mathrm{w}(A_v, v) \leq w(C_v) \tag{1}$$

holds for all active nodes v. Since the last two nodes s and t of A are in different components of the graph induced by C, t is an active node and therefore $\mathrm{w}(A_t, t) \leq \mathrm{w}(C_t) = \mathrm{w}(C)$ by inequality 1, i. e. the cut-of-the-phase has a lower weight than the arbitrary s-t-cut, so it is a minimal s-t-cut.

Inequality 1 is proven by induction on the set of active vertices. For the first active vertex v, the C_v includes exactly the edges that connect A_v and v, so $\mathrm{w}(A_v, v) = w(C_v)$.

It is now assumed that inequality 1 holds for all active vertices up to a node u, and u is the next active vertex. Then

$$\mathrm{w}(A_u, u) = \mathrm{w}(A_v, u) + \mathrm{w}(A_u \setminus A_v, v), \tag{2}$$

i. e. the sum of the edge weights can be decomposed into the edges that are connecting u with the nodes in A_v and those that are connecting it with the nodes in A_u that are not in A_v. Further, $\mathrm{w}(A_v, u) \leq \mathrm{w}(A_v, v)$ as v was the most tightly connected vertex when it was added to A. So by induction, $\mathrm{w}(A_v, v) \leq w(C_v)$, and with equation 2 the following holds:

$$\mathrm{w}(A_u, u) = \mathrm{w}(C_v) + \mathrm{w}(A_u \setminus A_v, u)$$

Since $C_v \subset C_u$, $E(A_u \setminus A_v) \subset C_u$ and $C_v \cap E(A_u \setminus A_v) = \emptyset$,

$$\mathrm{w}(A_u, u) = \mathrm{w}(C_v) + \mathrm{w}(A_u \setminus A_v, u) \leq \mathrm{w}(C_u),$$

holds as claimed. □

The lemma shows that each call of `MinimumCutPhase` correctly computes a minimal s-t-cut. Afterwards, s and t are merged which prevents the same cut to be computed twice and reduces the size of V by one in each iteration, so the algorithm terminates. The weights of edges from another node to s and t are added, edges between s and t are removed.

A min-cut of a graph is a minimal s-t-cut for some s and t. The algorithm computes all $|V| - 1$ different minimal s-t-cuts, so it finds and returns a min-cut of G.

4.4.3 Complexity

The introduced min-cut algorithm calls MinimumCutPhase until $|V| = 1$. Since the called function reduces $|V|$ by one as it merges exactly two nodes, this results in $|V| - 1$ calls. The weight of the returned min-cut is calculated during the function call and can be stored, so the complexity is in $\mathcal{O}(|V| \cdot f)$, where f is the complexity of the called function.

MinimumCutPhase appends all nodes to the list A, which means $|V|$ constant operations. However, the most tightly connected vertex has to be computed. This is done with a priority queue that sorts all vertices $v \in V \setminus A$ according to $\mathrm{w}(A, v)$. Extracting the maxium is a $\log |V|$ operation. Every time a vertex v is removed from the queue and added to A, for each edge incident to a v it has to be checked whether the other node incident to the edge is in the queue, and its weight has to be updated, which are constant operations in an appropriate data structure. Since each edge is incident to exactly two nodes, this results in a time requirement of $2|E|$. Since $\mathrm{w}(A, t)$ has already been computed for the last node t of A, it can be returned with all edges incident to t, so the overall complexity of this function is in $\mathcal{O}(|E| + |V| \log |V|)$.

The authors suggest *Fibonacci heaps* [FT87] as data structure for the priority queue.

4.5 `SplitDist`

The tool `SplitDist` [Mai] computes the split distance (see section 3.2.3) for two trees sharing the same leaf set:

```
sdist --print-norm <tree1> <tree2>
```

The input trees have to be specified in *Newick* notation. `-print-norm` prints the normalized split distances for `<tree1>` and `<tree2>` as well as for `<tree2>` and `<tree1>`, since the split distance is not symmetric.

This tool is used here to compare cotrees of the true cograph and the edited cograph generated by the heuristic solution that will be described later in this work. Note again that using the split distance for this task has some disadvantages as described in section 3.2.3.

4.6 `CographCompletion.jar`

The algorithm described in section 3.3.9 has been implemented as a *Java* application by Sebastian Siemoleit. As described in section 4.1, `coedit` gives up on prime modules that need more than k edition operations. And since k must be set to a value such that multiple prime modules can be edited in a short amount of time, choosing it too large will make the computation infeasible. In such cases, an alternative approach to prime module editing is required. The original idea was to use a min-cut to split up the prime module along low weight edges, however it turned out that due to their structure, which is close to that of a complete graph, this will not produce good results. The cograph completion is a more sensible solution since it tries to create a cograph, too, and its performance is good enough to apply it even to bigger graphs. Yet, one big disadvantage arises when using `CographCompletion.jar`: since it does not know anything about the semantics of orthology, it also connects genes from the same species, thus labeling them as orthologs which is obviously wrong. This shortcoming could be corrected in future work.

5 Heuristic for Editing Cographs Based on Orthology Data

The primary goal of this work is to develop a heuristic for the cograph editing problem on graphs generated from orthology data.

For a set of genes X, each belonging to exactly one species, the tool `POFF` is used to compute all pairs of orthologous genes. The input graph $G = (X, E)$ then represents the orthology relation, where E contains an edge connecting each ortholog pair. `POFF` runs the sequence comparison tool `blast` to compute a bitscore for each pair of genes (x, y). These are not symmetric, and therefore the mean value of the bitscores of (x, y) and (y, x) is used as weight $\mathrm{w}(\{x, y\})$ for each edge $\{x, y\}$.

According to the results of section 3.3.1, G should be a co-graph, however, due to the inevitable false positive and false negative detections during the reconstruction of the orthology relation, this is not the case. Therefore, G shall be modified by adding or deleting edges such that it becomes a cograph. This cograph then allows the reconstruction of the gene tree. Since cograph editing in general is NP-complete [Liu+11], a heuristic approach based on local editing and bitscores will be used to iteratively eliminate the $P4$s in G, generating a cograph from it.

5.1 Outline of `orthoDeprime`

The tool that iteratively modifies an input graph, representing an estimated orthology relation, until it is a cograph is called `orthoDeprime`. The algorithmic outline is shown in listing 2. A more graphical overview is given in figure 12.

The algorithm consists of a loop that is repeated until there are no more prime modules found in the input graph, i. e. it is a cograph. In this loop, at first the prime modules are computed. Then, `coedit` is called for each prime module and the "best" of the resulting edition operation sets is applied to the input graph. If for a certain prime module no edition set can be found, an inclusion minimal cograph completion set is computed instead. Should this attempt fail, too, an edge cut set is computed and applied as a last resort to be able to continue the computation.

5.2 Identifying and Eliminating the Prime Modules

`orthoDeprime` uses the function getPrimeModules() to find the prime modules in the input graph. It checks whether the graph is disconnected, and if it is, it continues recursively on each component, else the complement of the graph is tested. If it is connected, too, a prime module has been found.

As described in section 3.3, cographs can be decomposed into parallel and series modules. They do not contain any prime modules. But since the input graph is based on an estimation of the orthology relation, it is most likely not a cograph. However, since orthology detection tools like `POFF` produce good results, it can be assumed that the deviation from the cograph structure is not too high. Indeed, most

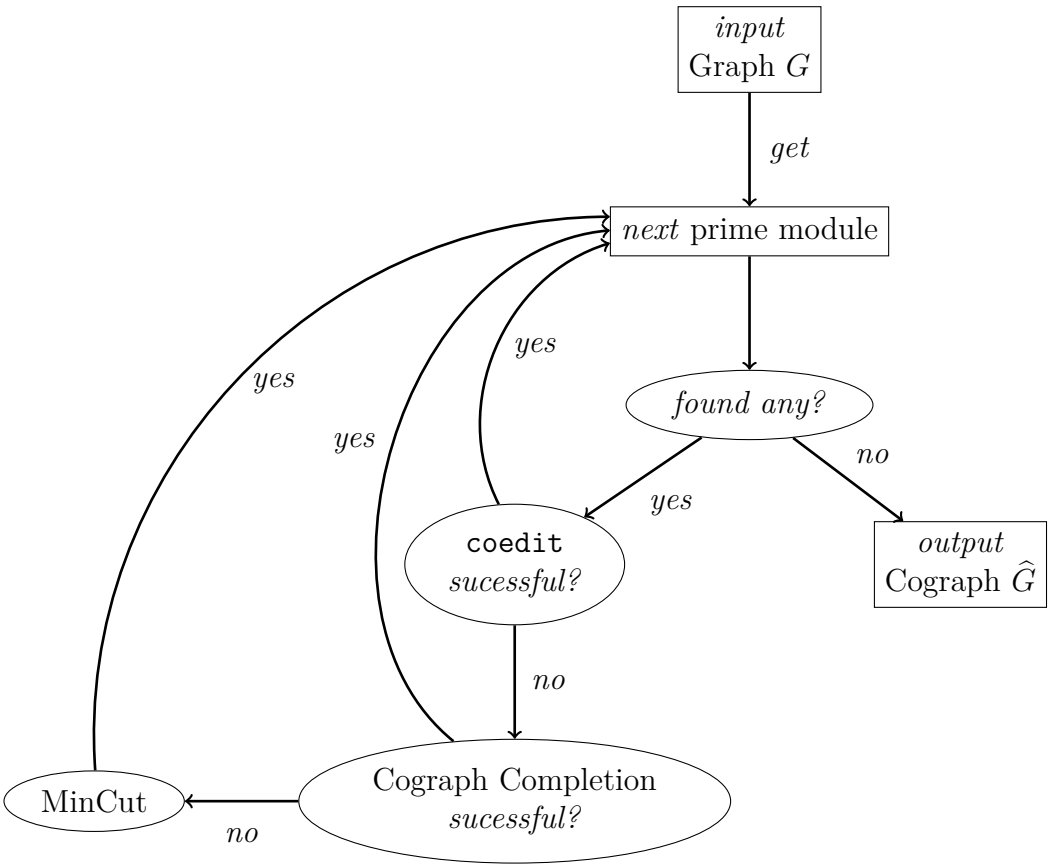


Figure 12: A schematic overview of `orthoDeprime`. One after the other, the algorithms for cograph editing, cograph completion and the min-cut are applied to each prime module in the graph. If any of them succeeds, the resulting edition set is applied to the current prime module and the next module is processed.

of the prime modules found in the orthology graphs constructed with the output of `POFF` usually have a size of ten to 20 vertices, see figure 17 for an analysis of the prime module sizes of a sample dataset. This is very low compared to the size of the graphs containing them. The prime modules are those parts of the graph that contain the induced P_4s, which are forbidden subgraphs for cographs. The idea is that editing them locally will also reduce the number of P_4s globally, though new P_4s could arise in the graph by editing the prime modules locally.

Once all prime modules in the graph are identified, the tool `coedit` (see section 4.1) is used on each of them to find an edge edition set, i. e. a set of edges to be removed or added from the graph, such that after applying it the prime module is converted to a cograph. The search for such an edition set, however, is not always successful, as described in the next section.

Algorithm 2: orthoDeprime()

Input: Input graph G
Output: Edited cograph $\widehat{G}$
repeat
  primes ← G.getPrimeModules();
  foreach prime *in* primes **do**
    run `coedit` on *prime* to compute cograph edition sets;
    eliminate edition sets connecting nodes belonging to the same species;
    bestSet ← choose edition set maximizing the average edge weight;
    if *no edition set found* **then**
      bestSet ← compute inclusion min cograph completion set for *prime*;
      if *no cograph completion set found* **then**
        bestSet ← compute a minimal edge cut set of *prime*;
    apply *bestSet* to G;
until *no more edition operations have been applied*;
return G;

5.3 Non-editable Prime Modules

Due to the exponential time complexity of the algorithm used by `coedit`, prime modules of a certain size exceed the computational limits and an edition set cannot be computed for them. In `coedit`, the maximal size of an edition set is bounded by a constant k which limits the complexity to polynomial time, and `orthoDeprime` uses a default value of $k \leq 6$.

If `coedit` should fail to find a valid edition set, `orthoDeprime` tries to compute an inclusion minimal cograph completion set using the tool `CographCompletion.jar` (see section 4.6). The algorithm used has a better time complexity and should be able to cope with the most prime modules in the input data even if they have a size of 30 to 40 vertices. This tool has one huge disadvantage though. Since it does not interpret the nodes as genes, it cannot distinguish different gene families and adds edges between genes from the same family, though by definition they cannot be orthologous. Adding this functionality to the cograph completion is a promising approach for future improvements of `orthoDeprime`.

To be able to continue processing the input graph if no edition set is found, the prime module is decomposed into two disjoint components by computing and applying a *minimal cut set*, i. e. a set of edges of minimal weight such that removing them disconnects the prime module into two components. This cut set is computed using the algorithm described in section 4.4 [SW97], which has a much lower time complexity than `coedit`. Once the prime module has been split into two components, each will be decomposed further in the next iteration or, if necessary, cut again until

Function getPrimeModules

Input: called on a graph G

Output: list of prime modules in graph G

$primes \leftarrow \emptyset$;

if $|V(G)| < 4$ **then** graph cannot contain a P_4
    return *primes*;

else if G *is disconnected* **then**
    foreach *component* comp *in* G **do**
        primes.append(*comp*.getPrimeModules());

else if $\overline{G}$ *is disconnected* **then**
    foreach *component* comp *in* $\overline{G}$ **do**
        primes.append(*comp*.getPrimeModules());

else
    primes.append(G);

return *primes*;

an edition set can be computed.

5.4 Choosing the Best Edition Set

Since `coedit` computes all possible edition sets with k or less edition operations, there in general multiple edition sets available for each prime module, from which one needs to be chosen. `orthoDeprime` chooses the set that, when applied to the prime module, maximizes its average edge weight. This ensures that, on the one hand, preferably edges with a high weight are added, but not more than necessary, and on the other hand, preferably edges with low weights are removed from the graph.

The `MinCut` algorithm chooses the minimal cut based on the lowest sum of edge weights, so it will also preferably remove edges with low weight, i. e. those with low confidence, from the graph.

Since the weights of edges which are not present in the input graph are unknown, `orthoDeprime` requires a resource to look them up. Therefore, for each run a subfolder `./blast/` containing *all vs all* comparisons and bitscores for any two genes from different species, is necessary. `POFF` already computes them using `blast`, and these results can be stored by adding the `-storebla` parameter (`-keep` in newer versions of `Proteinortho`):

```
poff.pl -project=<projectName> -storebla=1 <fastaFiles>
```

The parameter `-project` is a prefix `POFF` adds to the output files.

5.5 Implementation and Usage

`orthoDeprime` is implemented in *C++*. When running it, the definition files for the forbidden subgraphs for P_4-sparse graphs which are required by `coedit` need to reside in the same folder as the executable. Additionally, the tool `CographCompletion.jar` must be added to the `PATH` variable such that the operating system can locate it when it is called. Since `CographCompletion.jar` is written in *Java*, a *Java Runtime Environment* needs to be installed. The usage of `orthoDeprime` itself is explained in section 6.6.

5.6 A Small Example

In this section, a small example demonstrating the functionality of `orthoDeprime` is provided.

5.6.1 The Input Data

The graph from figure 13 serves as input data. It consists of three connected components: a *5-clique*, i. e. a maximal complete subgraph with five vertices, a large prime module consisting of 24 nodes and 167 edges, and a small prime module consisting of just five vertices and six edges. The components were extracted from a larger data set called `20_20` that is described in more detail in section 6.2. The labels of the nodes contain both the species name (*before* the underscore) and the gene name (*after* the underscore) of the gene each node represents. All edges are weighted with the `blast` bitscores resulting from an *all versus all* comparison of the genes, however, for a cleaner visualization, the edge weights are omitted in the figure.

5.6.2 Locating the Prime Modules

Running `orthoDeprime` on the described dataset, it will begin by applying a modular decomposition to the graph, but only keep the found prime modules. In this case, there are two prime modules, while the clique is a cograph since it does not contain an induced P_4. Then, the two prime modules are processed.

5.6.3 Processing the Small Prime Module

Assume that the small prime module is edited first. `orthoDeprime` calls `coedit` on this module. The parameter k of `coedit`, defining the maximal number of edges in the edition set, is initialized with two as default value. If no edition set of size two was found, it would be incremented by one and another attempt would be made to find a solution. This process is iterated up to a maximal value of five per default since higher values require too much computation time.

However, in this case, 13 edition sets with just one or two operation can be found. Since in the small module all genes are from different species (see the labels in figure 13), no edition sets are removed. Now, the set maximizing the average edge

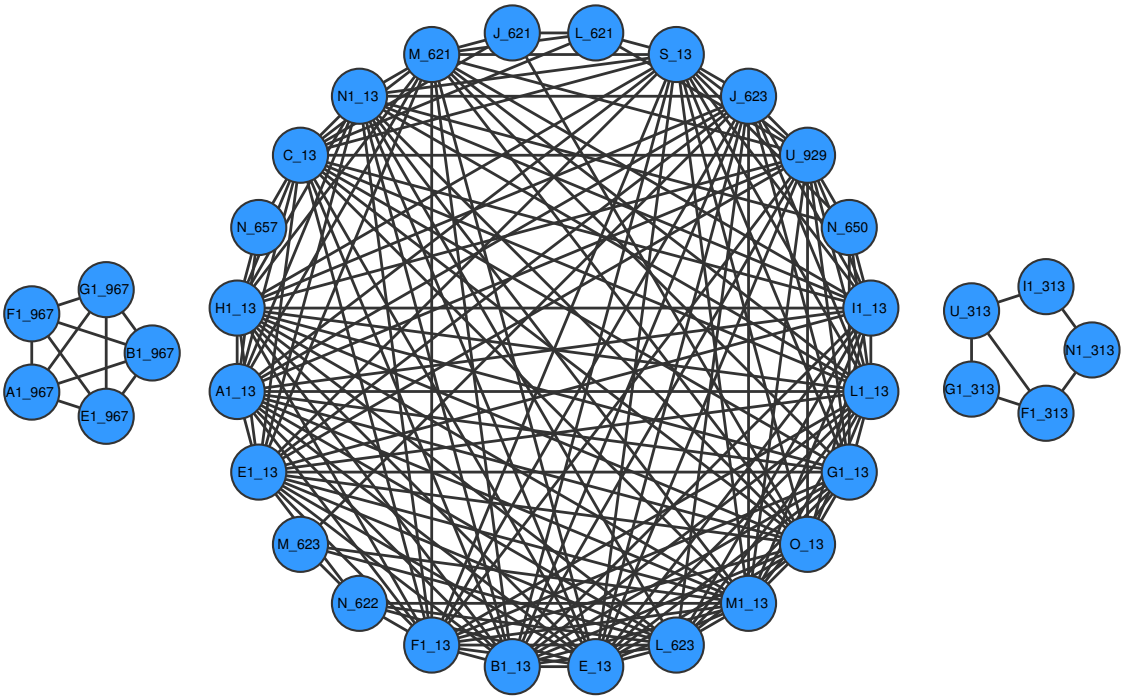


Figure 13: The input dataset, consisting of three components. Two of them are prime modules while the third one, located on the left, is a complete graph and therefore a cograph.

weight is determined. The weight of non-edges to be inserted is extracted from the `blast` results stored in the directory `./blast/`, if this is possible. The best edition set found is $E^+ = \{\{N1\_313, U\_313\}, \{F1\_313, I1\_313\}\}$ and $E^- = \emptyset$. It is then applied to the input graph.

5.6.4 Processing the Large Prime Module

The large prime module is treated like the small one, however, this time `coedit` fails to find any edition sets even for $k = 5$. Therefore, an alternative strategy has to be used: the tool `CographCompletion.jar` is called. Since it is a probabilistic algorithm, it will return different results every time it is used, but it could for instance return a cograph completion consisting of 148 edges. Since this prime module originally has 24 nodes and 167 edges, and a complete graph of this size has $\binom{24}{2} = 276$ edges, adding 148 edges results in an "almost" complete graph with $167 + 78 = 245$ edges.

5.6.5 Finishing

After all prime modules have been processed, `orthoDeprime` checks again whether there are any prime modules in the graph. Since none are found, it writes the edited graph to the output file and terminates. Figure 14 shows the result.

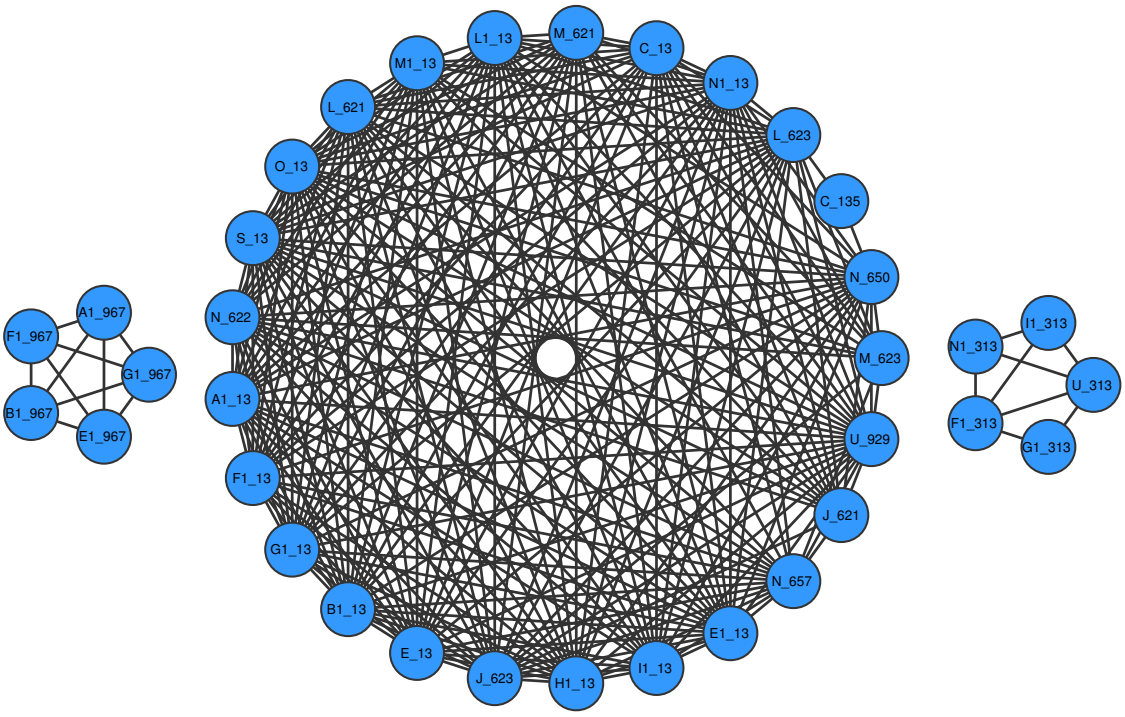


Figure 14: The edited output graph, which is now a cograph.

6 Results and Evaluation

In this section the results of applying `orthoDeprime` to two sample datasets will be shown. The goal is to find out whether the heuristic is capable of processing real world problem sizes and to compare the output to the input generated by `POFF` and the true orthology graph.

6.1 A Pipeline for Phylogenetic Tree Inference

Using the tools of other authors along with the programs developed in this work it is possible to construct a software pipeline that can infer a phylogenetic tree for a number of species from a given set of sequences of their genes or proteins, as shown in figure 15. Therefore, `POFF` is applied to the input sequences to estimate the orthology relation of the genes. The result is a list of weighted edges, which is converted to an adjacency list using the script `poffToAdjb`. Then, `orthoDeprime` can be used to restore the cograph property of the estimated orthology graph. The result is a cograph from which the cotree in *Newick* notation can easily be generated by the tool `cographToNewickCotree`. This tree is a labeled gene tree and a species tree can now be inferred [HR+12].

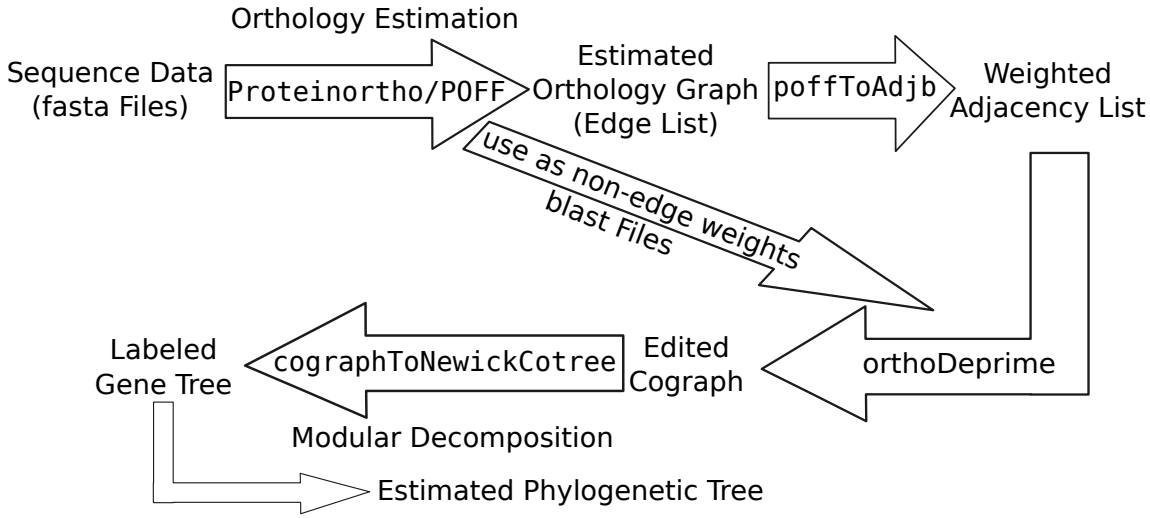


Figure 15: Overview over a possible pipeline for the prediction of phylogenetic trees. In this work, the inference of phylogenetic trees from gene trees was not explained, but there exist methods to solve this task [HR+12]

.

6.2 Tested Datasets

The tool `orthoDeprime` described in section 5 was tested mainly on two sets of computationally generated sequences: `20_20` and `20_100`, each consisting of 20 respectively 100 gene families in 20 distinct species. They are the result of a

simulated evolutionary scenario considering gene duplication, gene cluster duplication and gene loss as possible events. The set `20_20` represents 1,565 genes featuring 65,644 orthologous pairs, while `20_100` represents 27,258 genes featuring 411,471 orthologous pairs. Those datasets were generated by the authors of `POFF` [LHR+13] and can be obtained on its homepage. Figure 16 shows a visualization of the smaller `20_20` dataset generated using a force-based layout algorithm.

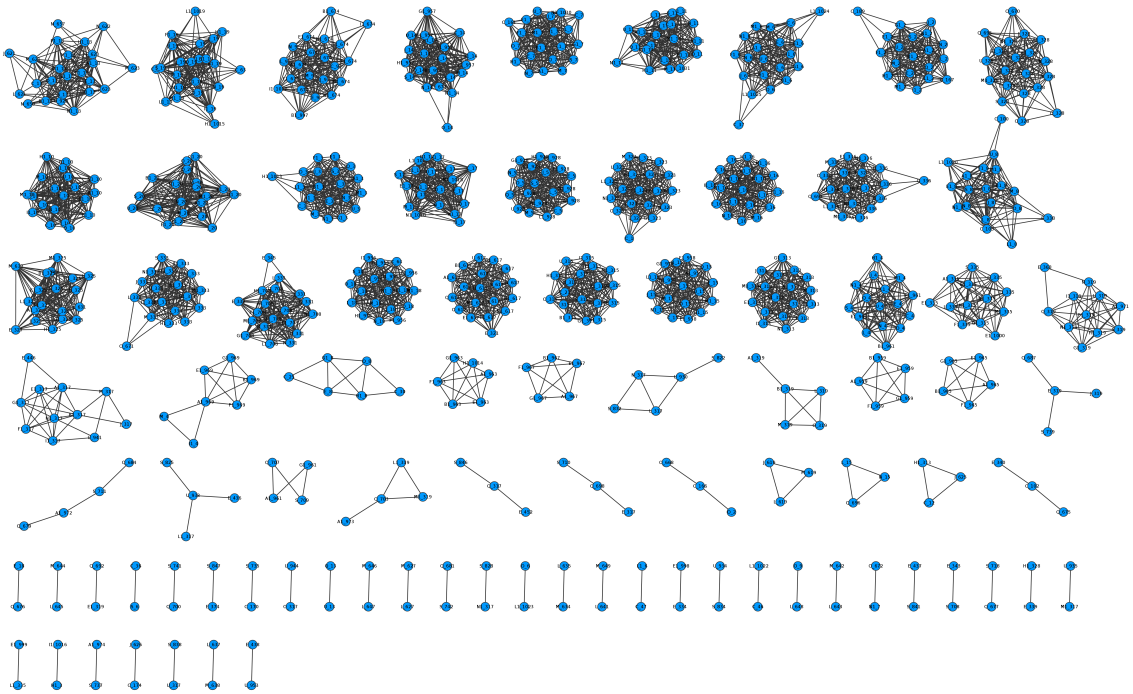


Figure 16: A visualization of the smaller `20_20` dataset generated using a force-based layout algorithm.

As described in [LHR+13], the datasets were simulated using the *Age Model* proposed by Keller-Schmidt et al. in 2010. This model relies on an initial species tree where the paths from the root to any leaf have length 1. Then, for the species tree, gene trees are simulated by using certain rules which independently modify the edges with a constant rate. Initially, an order list of ancestral genes is added to the root of the tree. From each gene, one gene family will evolve during the simulation. The user can decide on the number gene families. Then, the gene list is copied to both children of any internal node. Now, a *Poisson Process* is used to decide which and how many events, i. e. gene duplication, cluster duplication, genome duplication or gene loss, each havin a fixed probability of occurrence, are applied to the gene list. Multiple copies of a gene increase the chance that a copy of that gene is lost. Finally, a number of genes, sampled from a uniform distribution, are rearranged or inverted.

Additionally to the sequences, each dataset `20_<x>` contains a file (`m20_<x>`) with orthology matrices for each gene family, where a 1 in row i and column j means that the genes i and j are orthologs. From this file, the adjacency list of a graph

20_<x>.adj containing all genes and all orthology relations of each dataset is then constructed using the *Perl* script extractGraphFromMatrix:

```
extractGraphFromMatrix m20_<x> 20_<x>_true.adj
```

This graph is used to evaluate how good POFF predicts the true orthology relation based on the sequences and how the quality is affected by orthoDeprime. Note that, since m20_<x> truly represents an orthology relation, the resulting graph is a cograph (section 3.3.1), and therefore its gene tree can be obtained easily by applying modular decomposition (section 3.3.3).

The reason that simulated data was used for testing is that POFF requires synteny information for the input sequences to predict the orthology relation, and reliable biological data meeting this requirement is not available [LHR+13]. Another aspect is that for simulated data the true orthology relation is known with certainty. The downside is that one cannot be sure that the generated sequences are representative for true biological data.

6.3 Estimation of the Orthology Relation

The next step is to analyze the sequences from the given datasets and identify orthologous genes. As explained before, this is done by using POFF (see section 4.3). The latest version of Proteinortho (see section 4.2) integrates the functionality of POFF which is the reason why the most up-to-date version of Proteinortho with the -synteny switch is used for the orthology detection:

```
proteinortho5.pl -project=20_<x> -synteny -sim=<y>
    -keep ./seq/*.faa
```

This command instructs Proteinortho to detect all orthologous genes from the sequences stored in ./seq/. A similarity threshold of <y> for the reciprocal best hit heuristic is used and the result is stored in a file prefixed by the string following the -project switch. The option -keep will make the tool store the pairwise blast results. They should be moved to a directory called ./blast/ since orthoDeprime will look for them there to get a weight information for non-edges.

For this evaluation, all similarity thresholds $y \in \{0.25, 0.5, 0.75, 0.85, 0.95\}$ have been used.

6.4 Construction of the Input Graph

Before using the heuristic algorithm, an appropriate input graph needs to be constructed. From sequence data stored in *multifasta* files and synteny information (*gff* file), POFF generates a *syngraph* file, listing on each line an edge and its bitscores. Using the *Perl* script poffToAdjb (based on a script from Maribel Hernandez-Rosales), an adjacency list with bitscores as weights is generated from the syngraph file:

```
poffToAdjb -i <inputFile.syngraph> -o <outputFile.adjb>
```

The adjacency list has an easy syntax:

```
node1 | [adjacentNode1:edgeWeight adjacentNode2:edgeweight ...]
node2 | [adjacentNode3:edgeWeight adjacentNode4:edgeweight ...]
...
```

where square brackets mark optional elements.

6.5 Prime Modules in the Datasets

Since the success of the edition process heavily depends on the quality of the input datasets, it is interesting to study their prime modules as an indicator for the distance of the input data to the true orthology relation. The figures 17 and 18 show the number of vertices per prime module for the graphs generated from the datasets `20_20` and `20_100`, respectively. It is obvious that the number of vertices per prime module is constantly low, rarely exceeding 20 nodes per prime module with only few outliers. If the prime modules were much bigger, the approach used in this work could not succeed.

6.6 Running `orthoDeprime`

Running `orthoDeprime` is straightforward:

```
orthoDeprime <infile> [<outfile> -w -v]
```

The above command runs the tool with the adjacency list `<infile>` of an estimated orthology graph, usually the result of running `POFF`. The result is stored to the file `<outfile>` or, if not supplied, to the file `<infile>.edt`. The switch `-w` is used for weighted input graphs. The input graphs are usually weighted since, when using the script `poffToAdjb`, the bitscores are extracted and added to the graph as edge weights. They are important for choosing an optimal edition set. The `-v` switch turns on the verbose mode and prints out any applied edition operation.

6.7 A Distance Measure for Graphs

In the course of this work, graphs will be generated and modified. It is therefore interesting to quantify how different two graphs G and H are. For this purpose a symmetric, normalized distance measure d is used, defined as

$$d(G,H) = \frac{|(E(G) \triangle E(H)|}{|E(G) \cup E(H)|} = \frac{\Big|\big(E(G) \setminus E(H)\big) \cup \big(E(H) \setminus E(G)\big)\Big|}{|E(G) \cup E(H)|},$$

i. e. the distance of G and H is the number of edges that are either in $E(G)$ or in $E(H)$ (but not in both), divided by the total number of edges. Obviously, this quotient is zero if $E(G) = E(H)$, one if $E(G) \cap E(H) = \emptyset$ or somewhere in between in other cases. Thus, d is normalized. Also, $d(G,H) = d(H,G)$ holds since the union

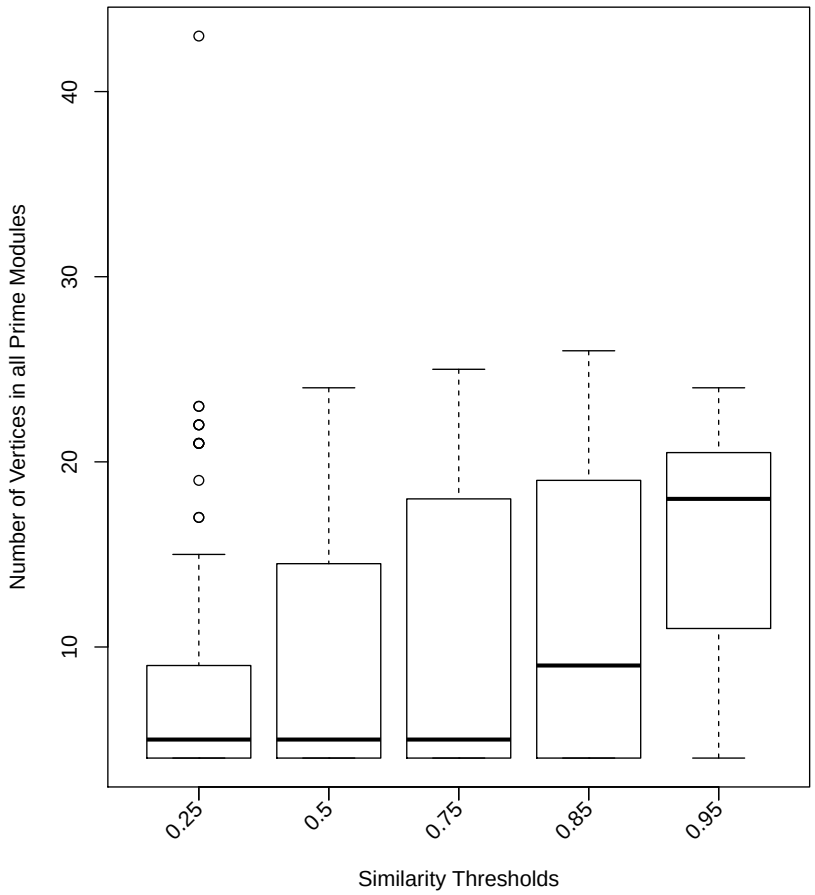


Figure 17: Boxplot of the number of nodes per prime module in each of the graphs constructed from data set `20_20` with the similarity threshold shown on the x-axis.

operation is commutative, so d is symmetric. Note that d does not take into account the number of vertices missing in G with respect to H or vice versa, but since in orthology graphs the edges carry the information, and since the input graphs used here do not contain isolated vertices (i. e. vertices that are not incident to any edge), it suffices to consider differences in the edge sets of G and H. Deleting a vertex in a graph that does not contain isolated vertices will always delete its incident edges and thus increase the distance anyway.

6.8 Evaluation of the Output

For the simulated datasets the true orthology graph G^* and an associated gene tree T^* which is identical to its cotree (see section 3.3.1) is known. Therefore, the output of `orthoDeprime`, the cograph $\widehat{G}^+$, can be compared to those and a distance score can be computed.

In an earlier attempt in the development of `orthoDeprime`, the cograph completion

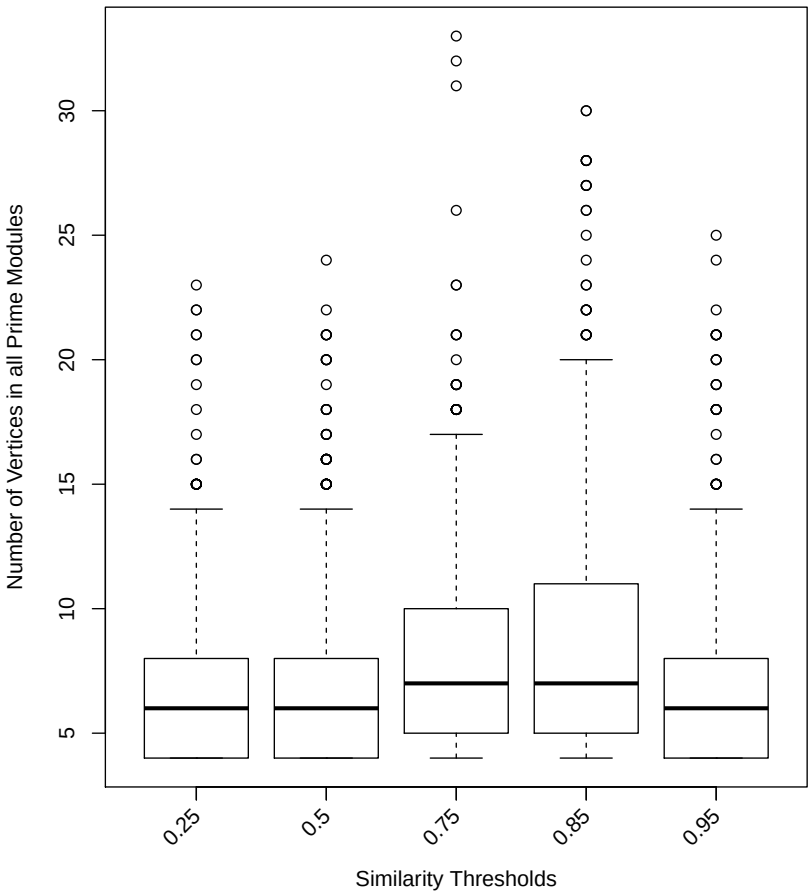


Figure 18: Boxplot of the number of nodes per prime module in each of the graphs constructed from data set `20_100` with the similarity threshold shown on the x-axis.

was not yet integrated. In this build, `orthoDeprime` applied `coedit` and, in case of failure, immediately used the min-cut procedure to decompose the graph. The resulting cograph $\widehat{G}$ has also been compared to the true graph, however, it could not improve the prediction of the orthology relation.

All stats and measures have been computed for both datasets, `20_20` and `20_100`, and for all graphs generated from them by varying the symmetry threshold of `POFF`. The results can be found in table 1 and table 2.

6.8.1 Number of Edges, Vertices and Prime Modules

Not surprisingly, the number of edges in the graphs decreases with an increasing similarity threshold for `POFF`s reciprocal best hit heuristic. For the default value `-sim=0.95`, G has only about as half as many vertices as the true graph G^*, and only about a tenth of the edges of the latter. Since `POFF` removes isolated nodes, the decreasing number of nodes is a logical consequence of the loss of edges. The

Table 1: Results of `orthoDeprime` for the dataset `20_20`. The input graph G of each column was obtained by running `Proteinortho v5c` with `POFF` enabled and the similarity parameter specified in the column header. $\widehat{G}$ is the edited graph, G^* is the true orthology graph, $\#_{primes}(G)$ is the number of prime modules in G, $d_{gr}(G, H)$ is the normalized symmetric graph distance of G and H, G^*_{pr} is the true graph G^* pruned to the vertex set of the input graph G, $d_{sp}(T_1, T_2)$ is the normalized split distance of the (co)trees T_1 and T_2, $\widehat{G}^+$ is the graph processed with cograph completion enabled editing.

graph G:	G^*	sim=0.25	sim=0.5	sim=0.75	sim=0.85	sim=0.95
$\|V(G)\|$	1,565	1,219	1,151	960	870	742
$\|E(G)\|$	65,644	5,046	5,031	4,886	4,842	4,753
$\#_{primes}(G)$	0	73	63	54	41	23
$\|E(\widehat{G})\|$	65,644	3,980	4,256	3,989	4,203	4,202
$\|E(\widehat{G}^+)\|$	65,644	5,883	5,613	5,338	5,302	5,048
$d_{gr}(G, \widehat{G})$	0	0.2395	0.1854	0.2034	0.1545	0.1406
$d_{gr}(G, \widehat{G}^+)$	0	0.1561	0.1158	0.1010	0.0957	0.0607
$d_{gr}(G, G^*)$	0	0.9353	0.9312	0.9307	0.9281	0.9285
$\|E(G^*_{pr})\|$	65,644	47,296	40,654	20,206	14,115	8,492
$d_{gr}(G, G^*_{pr})$	0	0.9105	0.8894	0.7774	0.6679	0.4505
$d_{gr}(\widehat{G}, G^*_{pr})$	0	0.9264	0.9051	0.8133	0.7111	0.5150
$d_{gr}(\widehat{G}^+, G^*_{pr})$	0	0.9048	0.8829	0.7658	0.6396	0.4204
$d_{sp}(\widehat{T}, T^*_{pr})$	0	0.3042	0.3107	0.2650	0.2526	0.2124
$d_{sp}(T^*_{pr}, \widehat{T})$	0	0.0634	0.0669	0.0772	0.0834	0.0934
$d_{sp}(\widehat{T}^+, T^*_{pr})$	0	0.3128	0.3203	0.2784	0.2648	0.2150
$d_{sp}(T^*_{pr}, \widehat{T}^+)$	0	0.0604	0.0645	0.0734	0.0792	0.0897

number of prime modules is an indicator for how close the structure of a graph is to the structure of a cograph. The tool `findPrimeModules` isolates and counts prime modules:

```
findPrimeModules <infile> [<outdir> -w]
```

The prime modules found in the graph `<infile>` are written to the directory `<outdir>`. As it can be observed in both datasets, increasing the similarity parameter results in a graph containing less prime modules (table 1 and table 2). An increased similarity threshold will cause only orthologs with a high sequence identity to be reported, thus recovering only parts of the true orthology graphs, however it seems in those highly conserved parts the cograph structure is preserved as well. This complies with the theoretical result that any induced subgraph of a cograph is again a cograph.

Table 2: Results of `orthoDeprime` for the dataset `20_100`. The input graph G of each column was obtained by running `Proteinortho v5c` with `POFF` enabled and the similarity parameter specified in the column header. $\widehat{G}$ is the edited graph, G^* is the true orthology graph, $\#_{primes}(G)$ is the number of prime modules in G, $d_{gr}(G, H)$ is the normalized symmetric graph distance of G and H, G^*_{pr} is the true graph G^* pruned to the vertex set of the input graph G, $\widehat{G}^+$ is the graph processed with cograph completion enabled editing. $d_{sp}(\widehat{T}, T^*_{pr})$ could not be computed because the memory demand of the computation of the cograph representation exceeded the available capacity.

graph G:	G^*	sim=0.25	sim=0.5	sim=0.75	sim=0.85	sim=0.95
$\|V(G)\|$	27,258	17,115	16,672	16,026	15,860	15,327
$\|E(G)\|$	411,471	40,351	40,739	42,973	46,615	42,979
$\#_{primes}(G)$	0	1,333	1,405	1,127	913	742
$\|E(\widehat{G})\|$	411,471	38,663	37,583	38,740	40,725	42,020
$\|E(\widehat{G}^+)\|$	411,471	43,967	45,683	50,213	55,906	45,355
$d_{gr}(G, \widehat{G})$	0	0.1180	0.1506	0.1634	0.1811	0.0664
$d_{gr}(G, \widehat{G}^+)$	0	0.1240	0.1527	0.1726	0.1807	0.0690
$d_{gr}(G, G^*)$	0	0.9395	0.9371	0.9236	0.9130	0.9056
$\|E(G^*_{pr})\|$	411,471	247,791	240,911	230,153	226,993	217,985
$d_{gr}(G, G^*_{pr})$	0	0.9017	0.8950	0.8661	0.8453	0.8233
$d_{gr}(\widehat{G}, G^*_{pr})$	0	0.9048	0.9006	0.8730	0.8544	0.8249
$d_{gr}(\widehat{G}^+, G^*_{pr})$	0	0.9012	0.8946	0.8660	0.8442	0.8208

6.8.2 Distance between the Input and Output Graphs

Comparing the input graph G generated from the orthology prediction of `POFF` with the output graphs $\widehat{G}$ (not using cograph completion) and $\widehat{G}^+$ (using cograph completion) of `orthoDeprime` by using the symmetric, normalized graph distance d_{gr} (see section 6.7) tells how much G needs to be changed to transform it to the output graphs.

Increasing the similarity threshold for `POFF` strongly decreases $d_{gr}(G, \widehat{G})$ in the dataset `20_20`. The distance reaches a maximal value of 24% for 25% similarity and is lowest with 14% for 95% similarity. In the `20_100` dataset the effect is different; here a higher similarity increases the distance up to a peak value of 18% at 85% of similarity, just to reach its minimal level with only 7% distance for 95% of similarity. This odd behavior does not seem to be significant since for $\widehat{G}^+$ the distance to G is constantly decreasing from 16% to 6% with increasing similarity. That is the expected behavior since for a higher similarity parameter, only highly conserved orthologs which are closer to the true graph are reported, making less editions necessary.

6.8.3 Distances and Edge Counts of the Pruned Graphs

In a next step, the graph distance of the output graph G of `POFF` to the true graph G^* is computed as well as the distances of the edited graphs $\widehat{G}$ and $\widehat{G}^+$ to G^*. However, this comparison is unfair since orthology detection tools like `POFF` are usually designed to report only candidates with high confidence to achieve a low *false discovery rate*, (i. e. maximize the number of true positives among all reported candidates), accepting that this results in a high number of false negative decisions.

To account for this problem, it is sensible to only consider the distance of that parts of the true graph G^* which have actually been reported by `POFF`. This can be achieved by *pruning* G^* to the vertex set of G.

The pruning is done with the tool `pruneGraph`:

```
pruneGraph <graph-to-prune> <pruneSetGraph>
    [<outfile> -wl -ws -cotlbl]
```

This tool extracts the vertex set V of the prune set graph. Then it prunes the graph-to-prune by removing any vertex that is not contained in V and all edges incident to one of those. The result is written to `<outfile>`. `pruneGraph` is applied to prune G^* with respect to $\widehat{G}$ to obtain the pruned graph G^*_{pr}. It can now be compared to G, $\widehat{G}$ and $\widehat{G}^+$.

The distances to the pruned graph G^*_{pr} are still very high. The modifications the old version of `orthoDeprime` performs on G^* result in a slightly higher distance (less than 1% more distant). This was a somewhat disappointing result since reconstructing the cograph structure was supposed to make the predictions of `POFF` more precise. The reason of this loss of similarity is probably that for a lot of larger prime modules, an adequate cograph edition set cannot be found due to computational limitations. In such cases the old build of `orthoDeprime` uses the min-cut algorithm to be able to continue the processing at all. Since many of the prime modules are, however, very dense and close to being a complete graph, the min-cut will likely remove only one vertex in each iteration until the size of the prime module is small enough to allow for an exact edition set to be found. This will cause many truly significant edges to be removed from $\widehat{G}$.

To fix this issue, `orthoDeprime` was updated to use the cograph completion algorithm on a non-editable prime module before it uses the min-cut. Since many prime modules are almost complete graphs, the cograph completion only adds a few edges and the result $\widehat{G}^+$ is closer to the true graph than the original prediction of `POFF`, though the improvements are very small in both datasets. Figure 19 visualized the distance of the edited graphs to the original one with respect to the chosen similarity threshold.

6.8.4 Computation of the Split Distances of the Cotrees

It is also interesting to analyze how different the gene trees $\widehat{T}$ and $\widehat{T}^+$ associated with the output graphs $\widehat{G}$ and $\widehat{G}^+$, respectively, are with respect to the true gene tree T^*.

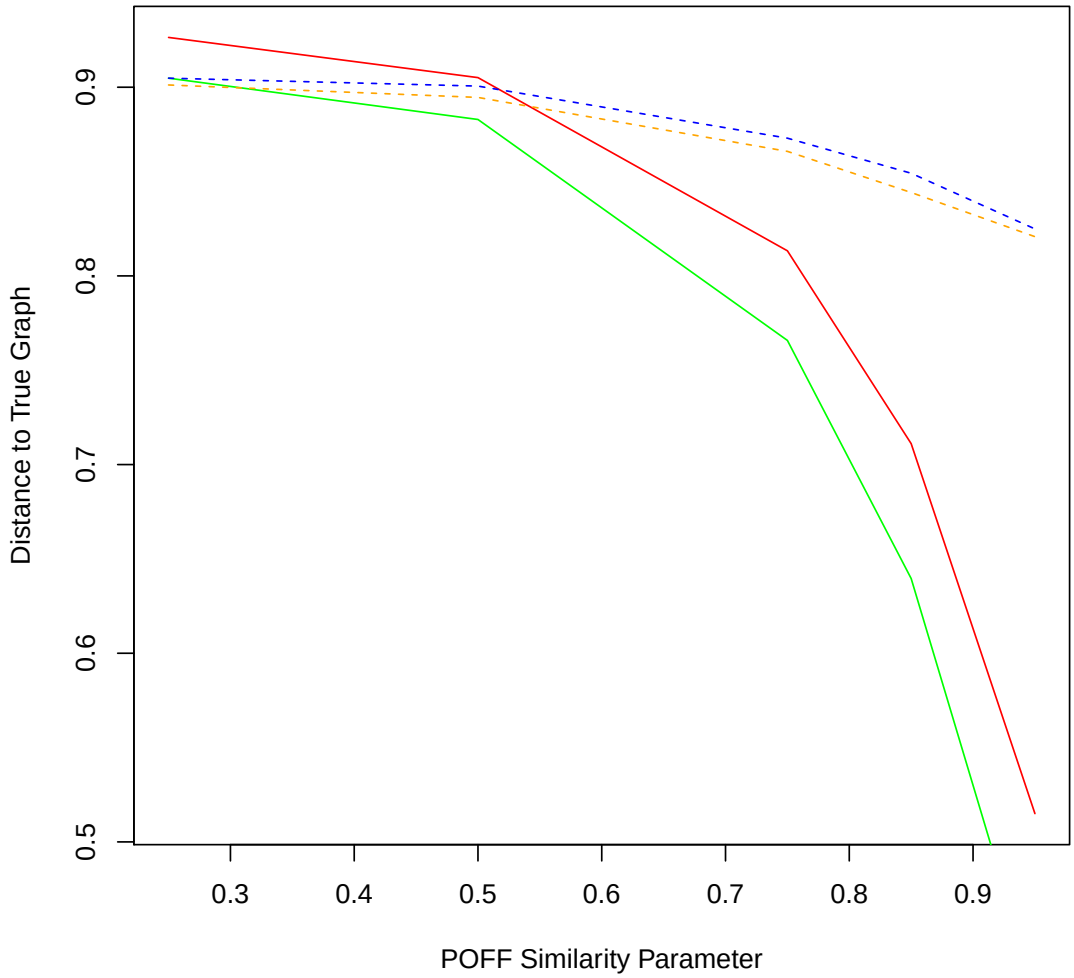


Figure 19: A visualization of the distances of the edited graphs $\widehat{G}$ (red and blue) and $\widehat{G}^+$ (green and orange) to the original one with respect to the chosen similarity threshold. The continuous lines mark the `20_20` dataset and the dashed ones the `20_100` dataset.

To do this, the gene trees belonging to $\widehat{G}$ and $\widehat{G}^+$ need to be constructed. As shown in section 3.3.1, those are exactly the cotrees of these graphs. How these can be computed is now shown for $\widehat{G}$, for $\widehat{G}^+$ the steps are analogous.

First, the tool `cographToNewickCotree` is used on $\widehat{G}$ to construct its cotree $\widehat{T}$ and store it in *Newick* notation (see section 3.2.2). The cotree is obtained by using modular decomposition (see section 3.3.3). The syntax is as follows:

```
cographToNewickCotree <infile.adjb> [<outfile.tree> -w -cotlbl]
```

The parameter `-w` is used for weighted input graphs as in this case, and `-cotlbl` enables the labeling of the internal nodes of the tree with 1 and 0 for series and parallel modules, respectively, marking as orthologous (paralogous, respectively) any two genes that have this internal node as their least common ancestor.

Now, the idea is to compare $\widehat{T}$ to T^* using the tree split distance (see section 3.2.3). The tool `SplitDist` (section 4.5) is used to compute them. However, to compute the distance, $\widehat{T}$ and T^* need to have the same set of leaves. Since `POFF` removes isolated vertices, correct edges not included in $\widehat{G}$ (false negatives) can lead to missing nodes and $\widehat{T}$ has a smaller leaf set than $T*$. The higher the `-sim` parameter is set for `POFF`, the more leaves will be missing, see e. g. table 1 for an overview of the number of leaves in the pruned graph for different similarity parameter choices.

The pruning is performed by applying `pruneGraph` to G^* with respect to $\widehat{G}$ to obtain the pruned graph G^*_{pr}, which is a cograph since G^* is a cograph and any induced subgraph of a cograph is again a cograph. Then `cographToNewickCotree` is used to generate its cotree T^*_{pr} and finally `SplitDist` can be used to compute the normalized split distances $d_{sp}(T^*_{pr}, \widehat{T})$ and $d_{sp}(\widehat{T}, T^*_{pr})$. Refer to section 4.5 for a usage description. Since $\widehat{G}$ and $\widehat{G}^+$ share the same vertex set, the cotree $\widehat{T}^+$ can also be compared to T^*_{pr}, and the distances $d_{sp}(T^*_{pr}, \widehat{T}^+)$ and $d_{sp}(\widehat{T}^+, T^*_{pr})$ can be computed.

6.8.5 Evaluating the Split Distances of the Cotrees

The results of the comparisons described above are listed in table 1 for the `20_20` dataset. Compared to the graph distance measure, the split distances are rather small. Interestingly, there are many splits in $\widehat{T}$ that are not in T^*_{pr} (distance ranges from 21% to 30%), while the inverse is not true (distance ranges from 6% to 9%). Additionally, $d_{sp}(\widehat{T}, T^*_{pr})$ is decreasing with an increasing similarity threshold since less erroneous edges are added to $\widehat{G}$ by `POFF`, while $d_{sp}(T^*_{pr}, \widehat{T})$ increases with increasing similarity due to the increasing amount of gene pairs rejected as orthologs.

For $\widehat{T}^+$, there are slightly more splits in $\widehat{T}^+$ that are not in T^*_{pr} than in in the tree generated without cograph completion, while there are slightly less splits in T^*_{pr} that are not in $\widehat{T}^+$. Again, the differences of about 1% are quite small.

Using the split distance, it was not possible to evaluate the `20_100` dataset since the construction of the cograph failed due to an insufficient memory capacity. This is caused by the huge edge count of the complemented subgraphs of the sparse input graph which are explicitly computed in the implementation used here. A more sophisticated implementation that is able to work without explicit computation of complements could solve this issue, but since the symmetric, normalized graph distance is available as an alternative measure this step was skipped in this work.

6.8.6 Consequences

In both datasets, the default symmetry parameter of 95% yields the best results both for the output of `POFF` and the edited graphs. A higher similarity also reduces the

number of prime modules and thereby speeds up the processing by `orthoDeprime`, so the default value is a good choice.

`orthoDeprime` has shown to slightly increase the accuracy of the orthology detection while at the same time reconstructing the cograph structure of the input graph. This is useful not only for the inference of gene trees or species trees, but also allows to use cograph-based algorithms for several common problems that are NP-hard on general graphs, but polynomial-time bounded on cographs, e. g. counting the number of cliques, computing the chromatic number or computing a generating formula for the set of cliques [CLB81].

7 Conclusion

This section concludes the results and achievements of this work. Additionally, it lists current limitations of the provided solution and gives an outlook on how these could be overcome in future approaches. Finally, it gives details on how the results of this work will be published.

7.1 Conclusion of the Results

In this work a heuristic approach to cograph editing on orthology graphs has been developed. The $C++$ implementation of the described algorithm is called `orthoDeprime` and has been tested with two simulated datasets to evaluate its quality. It has been shown that the edited graph is slightly less distant to the true graph than the original output of `POFF`, which means that the application of `orthoDeprime` improves the quality of the orthology prediction. At the same, the cograph property is restored such that existing tools can be used to infer a phylogenetic tree from the cotree of the edited graph, which is a labeled gene tree of the genes in that graph.

7.2 Limitations and Perspectives for Future Work

The idea of `orthoDeprime` is to iteratively apply edition operations to the input graph until it is a cograph. This is done by isolating prime modules in the graph and editing them locally. However, by successfully editing a prime module locally, new prime modules can arise on a global scale. Therefore there is no guaranty that this procedure will eventually terminate and output a cograph.

Another problem is that `CographCompletion.jar` possibly adds edges between genes from the same gene family, which is obviously wrong since those genes cannot be orthologous. A possible workaround would be to run the algorithm multiple times and hope that, due to its probabilistic character, it outputs a valid solution after a few tries. A better a approach would be to adapt the computation such that it never adds illegal edges in the first way. Fixing this issue is a promising way to achieve better results with `orthoDeprime`.

To which extend the problems mentioned above will arise strongly depends on the input data. If the input graph is "almost" a cograph, the processing will likely succeed with good results. In the context of orthology graphs, this means that a (at least partially) good estimation of the orthology relation, featuring a low false discovery rate, is required. This means that only orthologs with a high sequence identity can be detected, and thus the inferred gene tree will be incomplete. However, `Proteinortho` with the `POFF` extension has shown to be a reliable source of input data appropriate for `orthoDeprime`.

7.2.1 Publication of this Work

This thesis will be published via the document server[1] of the Faculty of Mathematics and Computer Science of the University of Leipzig.

All tools, their source code and the data files will be made publicly available on the homepage[2] of the Bioinformatics Group of the Institute of Computer Science, University of Leipzig. For questions, suggestions or bug reports, please contact the author[3].

[1] http://lips.informatik.uni-leipzig.de/
[2] http://www.bioinf.uni-leipzig.de/
[3] felix@bioinf.uni-leipzig.de

References

[Ber12] Sarah Berkemer. *Cograph Editing: An Approach to Adjust the Orthology Relation for the Reconstruction of Phylogenetic Trees (Bachelor's Thesis)*. Feb. 2012.

[CLB81] D.G. Corneil, H. Lerchs, and L.Stewart Burlingham. "Complement reducible graphs". In: *Discrete Applied Mathematics* 3.3 (1981), pp. 163–174. ISSN: 0166-218X. DOI: 10.1016/0166-218X(81)90013-5.

[Cor06] Denis Cornaz. "A linear programming formulation for the maximum complete multipartite subgraph problem". English. In: *Mathematical Programming* 105.2-3 (2006), pp. 329–344. ISSN: 0025-5610. DOI: 10.1007/s10107-005-0656-6. URL: http://dx.doi.org/10.1007/s10107-005-0656-6.

[DTS12] Daniel Doerr, Annelyse Thévenin, and Jens Stoye. "Gene family assignment-free comparative genomics". In: vol. 13. Proceedings of the Tenth Annual Research in Computational Molecular Biology (RECOMB) Satellite Workshop on Comparative Genomics. Niteroi, Brazil, 2012.

[FT87] M. L. Fredmanm and R. E. Tarjan. "Fibonacci heaps and their uses in improved network optimization algorithms". In: *Journal of the ACM* 35.3 (July 1987), pp. 596–615.

[GM95] S Guattery and GL Miller. "On the performance of spectral graph partitioning methods". In: *Proceedings of the sixth annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics. Philadelphia, PA, 1995, pp. 233–242.

[Hae66] Ernst Haeckel. *Generelle Morphologie der Organismen*. 1866.

[Hel+13] Marc Hellmuth et al. "Orthology relations, symbolic ultrametrics, and cographs". English. In: *Journal of Mathematical Biology* 66.1-2 (2013), pp. 399–420. ISSN: 0303-6812. DOI: 10.1007/s00285-012-0525-x. URL: http://dx.doi.org/10.1007/s00285-012-0525-x.

[HR+12] Maribel Hernandez-Rosales et al. "From event-labeled gene trees to species trees". In: *BMC Bioinformatics* 13.Suppl 19 (2012), S6. ISSN: 1471-2105. DOI: 10.1186/1471-2105-13-S19-S6. URL: http://www.biomedcentral.com/1471-2105/13/S19/S6.

[Lec+11] Marcus Lechner et al. "Proteinortho: Detection of (Co-)orthologs in large-scale analysis". In: *BMC Bioinformatics* (2011).

[LHR+13] Marcus Lechner, Maribel Hernandez-Rosales, et al. "Orthology Detection Combining Clustering and Synteny for Very Large Data Sets". 2013.

[Liu+11] Yunlong Liu et al. "Cograph Editing: Complexity and Parameterized Algorithms". In: *COCOON 2011*. Ed. by B. Fu and D.-Z. Du. LNCS 6842. Heidelberg: Springer Berlin, 2011, pp. 110–121.

[LMP08] Daniel Lokshtanov, Federico Mancini, and Charis Papadopoulos. “Characterizing and Computing Minimal Cograph Completions”. In: *Frontiers in Algorithmics*. Ed. by FrancoP. Preparata, Xiaodong Wu, and Jianping Yin. Vol. 5059. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 147–158. ISBN: 978-3-540-69310-9. DOI: 10.1007/978-3-540-69311-6_17. URL: http://dx.doi.org/10.1007/978-3-540-69311-6_17.

[Mai] Thomas Mailund. *SplitDist – Calculating Split-Distances for Sets of Trees*. University of Aarhus.

[Mou14] Mouagip. *genetic_code.svg*. Feb. 2014. URL: http://commons.wikimedia.org/wiki/File:Aminoacids_table.svg.

[SW97] Mechthild Stoer and Frank Wagner. “A Simple Min-Cut Algorithm”. In: *Journal of the ACM* 44.4 (July 1997), pp. 585–591.

[WS08] Kimitsuna Watanabe and Tsutomu Suzuki. “Universal Genetic Code and its Natural Variations”. In: *eLS* (Mar. 2008).

[Fak14] Prüfungsamt Fakultät für Mathemtik und Informatik. *Hinweise zum Anfertigen der Diplomarbeit (Bachelorarbeit, Masterarbeit)*. http://www.informatik.uni-leipzig.de/ifi/fileadmin/documents/studium/pruefamt/HINWEISE_Dipl_BSc_MSc_Arbeit_.pdf, Feb. 2014.

[The+12] The Encode Project Consortium et al. “An integrated encyclopedia of DNA elements in the human genome”. In: *Nature* 489 (Sept. 2012), pp. 57–74.