# Xenology Graph Completion

Thomas Gatter, Annachiara Korchmaros, Bruno J. Schmidt, and Peter F. Stadler

January 5, 2024

## 1 Introduction

Horizontal gene transfer (HGT) is a biological process where an organism's genome receives genetic material from sources other than its parental genomes. This process is a key mechanism in microbial evolution and is responsible for genetic innovation and the evolution of genome architecture. For instance, bacteria can acquire antibiotic resistance genes through HGT from other bacteria Soucy et al. [2015]. However, detecting HGT events from genomic data is still a challenging problem in computational biology, both mathematically and practically.

Usually, an event-labeled gene tree $T$ embedded in a species tree describes the evolution of a set of homologous genes (gene family history). The leaves of $T$ represent observable genes that evolve either vertically (through duplication or speciation events) or horizontally, through HGT events (as shown in Figure 1 on the left panel). According to Ravenhall et al. [2015], there are three main approaches to detecting HGTs: parametric methods that detect HGTs by measuring patterns correlated to the evolutionary history (e.g., compositional features Lawrence and Ochman [1997] and synteny index Adato et al. [2015], Sevillya et al. [2020]), phylogenetic explicit methods that first reconstruct a gene tree $T$ and a species tree $S$ of the originating species and then introduce HGTs to justify conflicts between them, and phylogenetic implicit methods where an unexpectedly short or long evolutionary distance from a given reference compared to the average can be caused by an HGT event Clarke et al. [2002], Dessimoz et al. [2008]. For a review of the tools available for inferring HGTs, please refer to Arnold et al. [2022].

Unfortunately, inferring HGTs is still challenging and often inaccurate, so it is natural to ask whether we can use a graph theory-based approach to reconstruct missing HGTs from partial information. Specifically, here we use the Fitch graph (xenology graph) associated with a gene tree $T$ to represent all the pairs of genes involved in an HGT (xenologous pairs) of $T$. Therefore, the problem of reconstructing missing HGTs is equivalent to build a Fitch graph that encodes the original xenologous information in 2-vertex subgraphs (xenology graph completion problem). This course shows two methods to reconstruct missing xenologous genes. The first method focuses on restoring full xenology relations from a partial, albeit reliable set of relations. The second method additionally maximizes a weighting function on the set of the missing relations. The weights in this case correspond to a certain confidence whether a relation (type) is present or not. In Section 2, we formally introduce Fitch graphs, overview recent results regarding the xenology graph completion problem, and outline a few heuristics to approach the problem. In Section 3, we will be listing the tasks for three different exercises (work packages) related to the dataset described in Section 4.

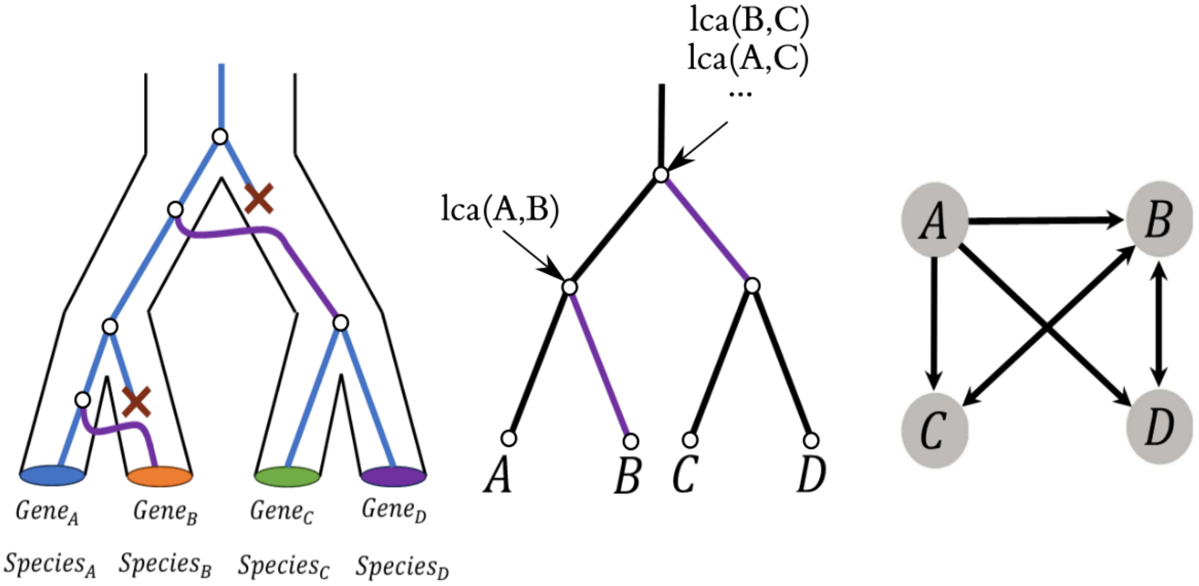

Figure 1: *Left panel:* A species tree $S$ (black outline) of four species with an embedded gene-tree $T$ (blue, purple). The horizontal gene transfer (HGT) edges in $T$ are highlighted in purple. Gene losses are annotated by red crosses. *Middle panel:* The embedded gene tree $T$ with deleted trimmed branches and contracted inner vertices with degree 2. The horizontal gene transfer edges are highlighted in purple. *Right panel:* The Fitch (xenology) graph $G$ associated to the tree in the middle panel. $(A, B), (A, C), (A, D)$ are edges in $G$ as the path from $\text{lca}(A, B)$ (resp. $\text{lca}(A, C), \text{lca}(A, D)$) to $B$ (resp. $C, D$) contains at least one HGT edge. The path from $\text{lca}(A, B)$ (resp. $\text{lca}(A, C), \text{lca}(A, D)$) to $A$ does not contain any HGT edges, hence $(B, A), (C, A), (D, A)$ are not edges in $G$.

# 2 Background

## 2.1 Fitch graphs

In this section, we summarize fundamental concepts and results about Fitch graphs. We denote by $V(G)$ and $E(G)$ the vertex-set and edge-set of any directed graph $G$, respectively.

**Definition 1.** Let $(T, \lambda)$ be a rooted tree with leaf-set $L(T)$ an edge-labeling $\lambda\colon F \to \{0, 1\}$. The *Fitch graph* associated to $(T, \lambda)$ is the digraph $G(T, \lambda)$ on $L(T)$ containing a directed edge $(x, y)$ for $x, y \in L(T)$ if and only if the (unique) path from $\text{lca}_T(x, y)$ to $y$ contains at least one edge $e \in F$ with label $\lambda(e) = 1$.

Fitch graphs are directed cographs (di-cographs), in particular. Every di-cograph $G$ is explained by a *cotree* $(T, t)$ with leaf set $L(T) = V(G)$, and an edge-labeling function

$t : V(T) \setminus L(T) \to \{0, 1, \overrightarrow{1}\}$ that uniquely determines the edge-set $E(G) = E_1(T,t) \cup E_{\overrightarrow{1}}(T,t)$ and the set of non-adjacent pairs of vertices $E_0(T,t)$ of $G$ where

$$\begin{aligned}
E_1(T,t) &:= \{(x,y) \mid t(\mathrm{lca}(x,y)) = 1\}, \\
E_0(T,t) &:= \{(x,y) \mid t(\mathrm{lca}(x,y)) = 0\}, \text{ and} \\
E_{\overrightarrow{1}}(T,t) &:= \{(x,y) \mid t(\mathrm{lca}(x,y)) = \overrightarrow{1} \text{ and } x \text{ is left of } y \text{ in } T\}.
\end{aligned}$$

In particular, any cotree explaining a Fitch cograph satisfies the following conditions.

**Definition 2.** A cotree $(T,t)$ is a *Fitch-cotree* if

(i) no vertex with label 0 has a descendant with label 1 or $\overrightarrow{1}$;

(ii) if a vertex $v$ has label $\overrightarrow{1}$, then the subtree $T(u)$ rooted at a $u$ child of $v$ does not contain any vertices with label 1 except the right-most child of $v$.

The forbidden structures of a Fitch-cotree are summarized in Figure 2. An example of a Fitch-cotree together with its associated Fitch graph is in Figure 3.

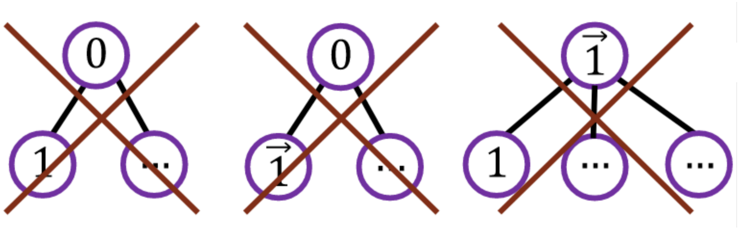

Figure 2: Forbidden structures in a Fitch-cotree. Vertices labelled 0 cannot be succeeded by any vertex labelled 1 or $\overrightarrow{1}$ (left panel and middle panel). Vertices labelled $\overrightarrow{1}$ cannot be succeeded by any vertex labelled 1 except for its rightmost child (right panel).

Hellmuth et al. Hellmuth and Scholz [2023] and Geiss et al. Geiß et al. [2018b] characterized Fitch graphs in terms of forbidden graphs or Fitch-cotrees.

**Theorem 2.1.** *For every digraph $G = (V,E)$, the following statements are equivalent.*

1. *$G$ is a Fitch graph.*

2. *$G$ does not contain an induced $F_1, F_2, \ldots, F_8$; see Figure 4.*

3. *$G$ is a di-cograph that does not contain an induced $F_1$, $F_5$ and $F_8$; see Figure 4.*

4. *$G$ is a di-cograph that is explained by a Fitch-cotree.*

5. *Every induced subgraph of $G$ is a Fitch graph.*

It is possible to transform a Fitch-cotree explaing $G$ into an edge-labeled tree that explains $G$ in $O(|V(T)|)$ time, avoiding the construction of the di-cograph altogether, using the procedure **cotree2fitchtree** described in Geiß et al. [2018a].

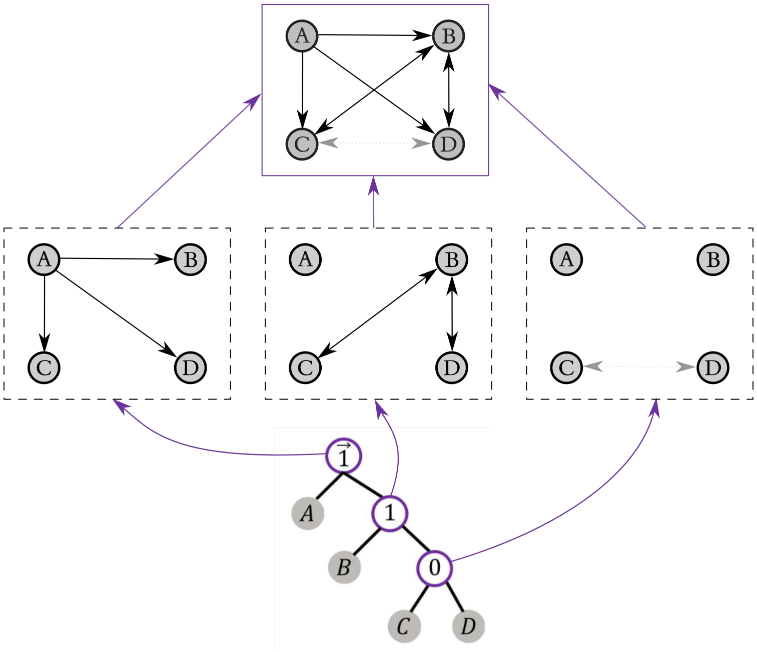

Figure 3: A Fitch-cotree $(T,t)$ (bottom panel) explaining the Fitch-grah $G$ (top panel). Black arrows represent the edges that each inner node of $(T,t)$ encodes, dotted edges between $C$ and $D$ are non-edges in $G$.

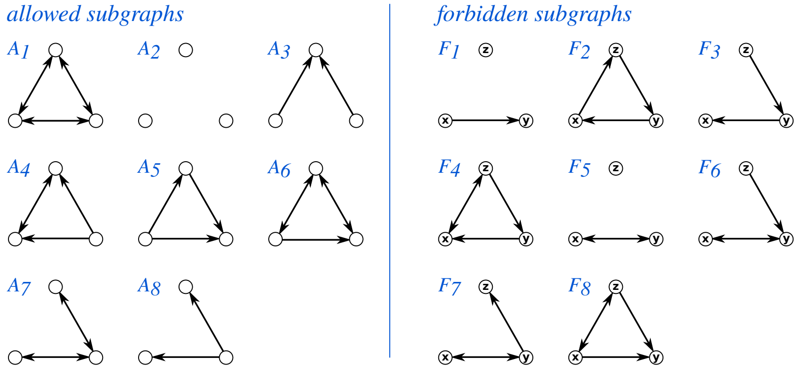

Figure 4: Of the 16 possible irreflexive binary relations on three vertices, eight ($A_1$ through $A_8$) may appear in Fitch graphs, while the remaining eight ($F_1$ through $F_8$) form forbidden induced subgraphs

## 2.2 Fitch-sat tuples of relations

A *binary irreflexive relation* $R$ on a set $V$ is a set of ordered pairs $(x,y)$ of elements of $V$ such that $x \neq y$ for all $x, y \in V$. The symmetric extension of $R$ is $\bar{R} := R \cup \{(x,y) \,|\, (y,x) \in R\}$. Example of binary irreflexive relation is the edge-set $E$ of any directed graph. Here, we mainly consider the (binary irreflexive) relations on the vertex-set $V$ of a Fitch graph associated with a gene tree. A binary relation $\ll$ is a *total order* if it is reflexive, transitive, antisymmetric and total. Let $G$ be a digraph, a total order $\ll$ on $V(G)$ is a *topological order* if $v \ll w$ whenever $(v,w) \in E(G)$. $G$ has a topological order if and only if $G$ is a DAG (directed acyclic graph). The effort to check whether $G$ admits a topological order $\ll$ and, if so, to compute $\ll$ is linear. Let $\mathcal{C} := \{C_1, \dots, C_k\}$ be a partition of $V(G)$ with $k \geq 1$, the *quotient graph* $G/\mathcal{C}$ has vertex-set $\{V_1, \dots, V_k\}$ and $V_i \to V_j$ for $i \neq j$ if there exists $x_i \in V_i$ and $x_j \in V_j$ where $(x_i, y_j) \in E(G)$. If $C_1, \dots, C_k$, $k \geq 1$ are the strongly connected components of a digraph, then $G/\mathcal{C}$ is a DAG.

We refer to a *tuple* as the triple of relations $\mathcal{E} = (E_0, E_1, E_{\overrightarrow{1}})$ on a set $V$ such that $E_0$ and $E_1$ are symmetric and $E_{\overrightarrow{1}}$ is antisymmetric. The tuple $\mathcal{E}$ is *full* if $E_0 \cup E_1 \cup E_{\overrightarrow{1}} = \{(x,y)|x, y \in V, x \neq y\}$, and it is *partial* if $E_0 \cup E_1 \cup E_{\overrightarrow{1}} \subseteq \{(x,y)|x, y \in V, x \neq y\}$. We say that $\mathcal{E}^* = (E_0^*, E_1^*, E_{\overrightarrow{1}}^*)$ *extends* $\mathcal{E}$ if $E_i^* \subseteq E_i$ for $i \in \{0, 1, \overrightarrow{1}\}$. Let $W \subseteq V$ and $R$ a set of relations, the set of *induced subrelations* by $W$ is $R[W] := \{(x,y) \in R \,|\, x, y \in W\}$. Let $\mathcal{E} = (E_0, E_1, E_{\overrightarrow{1}})$ be a tuple, the *induced tuple* by $W$ is $\mathcal{E}[W] := (E_0[W], E_1[W], E_{\overrightarrow{1}}[W])$.

At every Fitch-cotree we can associate a tuple of relations in the following way.

**Definition 3.** A tuple $\mathcal{E} = (E_0, E_1, E_{\overrightarrow{1}})$ is explained by a Fitch-cotree $(T,t)$ if $E_0 = E_0(T,t), E_1 = E_1(T,t)$, and $E_{\overrightarrow{1}} = E_{\overrightarrow{1}}(T,t)$. The tuple $\mathcal{E}$ is *Fitch*-satisfiable (in short *Fitch-sat*), if there is a full tuple $\mathcal{E}^* = (E_0^*, E_1^*, E_{\overrightarrow{1}}^*)$ that extends $\mathcal{E}$ and is explained by a Fitch-cotree $(T,t)$.

Moreover, a tuple $\mathcal{E}$ on $V$ is *Fitch*-satisfiable if it can be extended to $\mathcal{E}^*$ for which $(V, E_1^* \cup E_{\overrightarrow{1}}^*)$ is Fitch.

**Theorem 2.2.** *The partial tuple* $\mathcal{E} = (E_0, E_1, E_{\overrightarrow{1}})$ *on* $V$ *is* Fitch-sat *if and only if at least one of the following statements holds.*

(S1) $G_0 \coloneqq (V, E_1 \cup E_{\overrightarrow{1}})$ *is edge-less.*

(S2) (a) $G_1 \coloneqq (V, E_0 \cup E_{\overrightarrow{1}})$ is disconnected and
  (b) $\mathcal{E}[C]$ is *Fitch-sat* for all connected components $C$ of $G_1$

(S3) (a)(I) $G_{\overrightarrow{1}} \coloneqq (V, E_0 \cup E_1 \cup E_{\overrightarrow{1}})$ contains $k > 1$ strongly connected components $C_1, \dots, C_k$ collected in $\mathcal{C}$ and
  (II) there is a $C \in \mathcal{C}$ for which the following conditions are satisfied:
    (i) $G_0[C]$ is edge-less.
    (ii) $C$ is $\ll$-minimal for some topological order $\ll$ on $G_{\overrightarrow{1}}/\{C_1, C_2, \dots, C_k\}$.
  (b) $\mathcal{E}[V \setminus C]$ is *Fitch-sat*.

Theorem 2.2 provides us with a method for testing whether a partial tuple $\mathcal{E}$ is *Fitch-sat* by checking if any one of the three conditions (S1), (S2), or (S3) holds; this suggests a first *Fitch-sat* recognition Algorithm 1. In Algorithm 1, the *Fitch-sat* conditions are checked in a specific order, i.e., $(S_1, S_2, S_3)$; however, this order is not unique. Indeed, there are 5 more ways to recognize and eventually complete a partial tuple, namely the ordered triples $(S_2, S_1, S_3), (S_3, S_1, S_2), (S_3, S_2, S_1), (S_1, S_3, S_2), (S_2, S_3, S_1)$, which give a non-unique completion of a partial tuple. The following theorem establishes the correctness of Algorithm 1.

**Theorem 2.3.** *Let* $\mathcal{E} = (E_0, E_1, E_{\overrightarrow{1}})$ *be a partial tuple,* $n = |V|$ *and* $m = |E_0 \cup E_1 \cup E_{\overrightarrow{1}}|$. *Then, Algorithm 1 computes a Fitch-cotree* $(T,t)$ *that explains* $\mathcal{E}$ *or identifies that* $\mathcal{E}$ *is not* Fitch-sat *in* $O(n^2 + nm)$ *time.*

**Algorithm 1** Recognition of *Fitch-sat* partial tuple $\mathcal{E}$ on $V$ and reconstruction of a cotree $(T,t)$ that explains $\mathcal{E}$.

```
Input: Partial tuples $\mathcal{E} = (E_0, E_1, E_{\overrightarrow{1}})$
Output: A cotree $(T,t)$ that explains $\mathcal{E}$, if one exists or the statement "$\mathcal{E}$ is not Fitch-satisfiable"

Call BuildFitchCotree($V, \mathcal{E}$)
function BuildFitchCotree($V, \mathcal{E} = (E_0, E_1, E_{\overrightarrow{1}})$)
                    ▷ $G_0$, $G_1$ and $G_{\overrightarrow{1}}$ are defined as in Thm. 2.2 for given $\mathcal{E}$
    if $|V| = 1$ then return the cotree $((V, \emptyset), \emptyset)$
    else if $G_0$ is edge-less (otherwise if $G_1$ is disconnected) then
                    ▷ check (S1) (resp., (S2))
        $\mathcal{C} \coloneqq$ the set of connected components $\{C_1, \dots, C_k\}$ of $G_0$ (resp. $G_1$)
        $\mathcal{T} \coloneqq$ set $\{$BuildFitchCotree($C_i$, $\mathcal{E}[C_i]$) $\mid C_i \in \mathcal{C}\}$
        return the cotree from joining the cotrees in $\mathcal{T}$ under a new root labeled 0 (resp. 1)
    else if $G_{\overrightarrow{1}}$ has more than one strongly connected component then
                    ▷ check (S3)
        $\mathcal{C} \coloneqq$ the set of strongly connected components $\{C_1, \dots, C_k\}$ of $G_{\overrightarrow{1}}$
        $I \leftarrow \emptyset$
        for all $i \in \{1, \dots, k\}$ do
            if $G_0[C_i]$ is edge-less then $I \leftarrow I \cup \{i\}$
        if $I = \emptyset$ then
            Halt and output: "$\mathcal{E}$ is not Fitch-satisfiable"      ▷ (S3.a.II.i) not satisfied
        else if there is no topological order $\ll$ of $G_{\overrightarrow{1}}/\{C_1, \dots, C_k\}$ with $\ll$-minimal element $C_i$ with
        $i \in I$ then
            Halt and output: "$\mathcal{E}$ is not Fitch-satisfiable"      ▷ (S3.a.II.ii) not satisfied
        else
            $\ll \coloneqq$ a topol. order on the quotient $G_{\overrightarrow{1}}/\{C_1, \dots, C_k\}$ with $\ll$-minimal element $C^* \coloneqq C_i$ for
            some $i \in I$
            $\mathcal{T} \coloneqq \{$BuildFitchCotree($C$, $\mathcal{E}[C]$) $\mid C \in \{C^*, V \setminus C^*\}\}$
            return the cotree $(T,t)$ obtained by joining the cotrees in $\mathcal{T}$ under a new root with label $\overrightarrow{1}$,
            where the tree $(T^*, t^*)$ that explains $C^*$ is placed left of the tree $(\widehat{T}, \widehat{t})$ that explains $V \setminus C$
    else
        Halt and output: "$\mathcal{E}$ is not Fitch-satisfiable"
```

## 2.3 Weighted Fitch graph completion problem

In the contest of gene family histories, partial tuples represent partial xenologous information, which in practice are obtained from a first estimate of HGTs from sequence data (i.e., HGTs, with a significant p-value, obtained from implicit phylogenetic methods and/or parametric methods). Therefore, Algorithm 1 provides a first method to complete the xenologous information. However, this approach does not take into account that for every pair of genes the likelihood of having or not an HGT (e.g., sum of log-ratios of p-values obtained from different HGT-estimation methods) is also available. This observation suggests a second method to complete the xenologous information by maximizing a score on the non-classified pairs of genes.

For a partial tuple $\mathcal{E}$ on $V$, $\bar{\mathcal{E}} \coloneqq \{\{x,y\} \notin (E_0 \cup E_1 \cup E_{\overrightarrow{1}})^{sym}\}$ denotes the set of non-classified pairs of elements (genes). For every pair of vertices there are four possibilities $x :: y \in \{x \rightleftharpoons y, x \rightarrow y, x \leftarrow y, x \smile y\}$. In terms of xenologous relations, they correspond to having an HGT from $x$ to $y$ ($x \rightarrow y$), from $y$ to $x$ ($y \rightarrow x$), from $x$ to $y$ and viceversa ($x \rightleftharpoons y$), and not having any HGT ($x \text{\textvisiblespace} y$). At each of these cases we assign a score $w(x :: y)$ which corresponds to the log-odds ratio for observing one of the four possible xenology relationship as determined from experimental data.

Let $\mathcal{E}$ be a partial tuple on $V$, and let $G$ be a Fitch graph explaining $\mathcal{E}$. The score (weight) $w(x :: y)$ can be naturally extended to a score on the 2-vertex induced subgraph of $G$ by $x :: y$. The *score* of $G$ is the sum $f(G) = \sum_{\{x,y\} \in \bar{\mathcal{E}}} w(G[\{x, y\}])$. Given $\mathcal{E}$ and a scoring function on $\bar{\mathcal{E}}$, the *weighted Fitch Graph completion problem* asks for a Fitch-sat extension of $\mathcal{E}$ which maximizes the score of the associated Fitch graph.

The weighted Fitch-graph completion problem can also be seen as special case of the problem with empty tuple $\mathcal{E}^{\emptyset} := (\emptyset, \emptyset, \emptyset)$. To see this, we extend the weight function to all pairs of vertices by setting, for all input pairs, $w(G[\{x, y\}]) = m_0$ and $w(x :: y) = -m_0$ for $(x :: y) \neq G[\{x, y\}]$, where $m_0 \gg |V|^2 \max_{::,\{x,y\} \in \bar{\mathcal{E}}} |w(x :: y)|$. The choice of weights ensures that any Fitch graph $G'$ maximizing $f(G')$ induces $G'[\{x, y\}] = G[\{x, y\}]$ for all pairs $\{x, y\}$ in the input tuple, because not choosing $G[\{x, y\}]$ reduces the score by $2m_0$ while the total score of all pairs not specified in the input is smaller than $m_0$.

The weighted Fitch-graph completion problem is NP-complete  Hellmuth et al. [2023]. Therefore, ad-hoc heuristics must be designed to find an approximate solution. A simple greedy approach is summarized in Algorithm 2 and checks whether the additional edges produced one of eight forbidden subgraphs in Figure 4.

**Algorithm 2** Greedy to compute full $\mathcal{E}$ that is *Fitch-sat*.

```
Input: $V$ and set of weights $W(x, y) = \{w[x \rightleftharpoons y], w[x \rightarrow y], w[x \leftarrow y], w[x \text{\textvisiblespace} y]\}$ for all
        distinct $x, y \in V$.
Output: A full Fitch-sat set $\mathcal{E} = (E_0, E_1, E_{\overrightarrow{1}})$.
$E_0 := \emptyset$, $E_1 := \emptyset$, $E_{\overrightarrow{1}} := \emptyset$
index($\rightleftharpoons$) := 1 index($\text{\textvisiblespace}$) := 0, index($\rightarrow$) = index($\leftarrow$) := $\overrightarrow{1}$
$\mathfrak{W}$ := ordered set consisting of all $w[x :: y]$ for all $x, y \in V$ and $:: \in \{\rightleftharpoons, \rightarrow, \leftarrow, \text{\textvisiblespace}\}$ in
    non-decreasing order
for all $w[x :: y] \in \mathfrak{W}$ in non-decreasing order do
    if $\mathcal{E}$ where $E_{\text{index}(::)}$ is temporarily replaced by $E_{\text{index}(::)} \cup (x :: y)$ is Fitch-sat then
        replace $E_{\text{index}(::)} \in \mathcal{E}$ by $E_{\text{index}(::)} \cup (x :: y)$.
        remove $w[x :: y]$ from $\mathfrak{W}$ for all $:: \in \{\rightleftharpoons, \rightarrow, \leftarrow, \text{\textvisiblespace}\}$
return $\mathcal{E}$
```

# 3 Workpackages

## 3.1 WP0

- Familiarize yourself with the definitions presented in this project description.
- Download the dataset and our provided functions from `https://github.com/bsfaqu/fitch-graph-prak`. Ensure that the examples given in the `__main__` run without errors and make sure you understand all parts of the README, or Section 5.

## 3.2 WP1

The scope of this workpackage is to recognize whether partial tuple of xenologous relations is Fitch-sat, and compare the performances of different methods using the following steps.

1. Familiarize yourself with the dataset of xenology graphs in Section 4.

2. Implement a function that creates partial tuples $\mathcal{E} = (E_0, E_1, E_{\overrightarrow{1}})$ by deleting $10, 20, \ldots,$ 90% of the information in the xenology graphs (respecting the symmetry of the relations in $E_0, E_1$).

   - randomly remove from the set $(x, y) \in E_0 \cup E_1 \cup E_{\overrightarrow{1}}$
   - remove also $(y, x)$ if $(x, y) \in E_0 \cup E_1$

2b. add a special scenario where all elements of $E_{\overrightarrow{1}}$ are removed, hence $E_{\overrightarrow{1}} = \emptyset$, and then randomly remove from $E_0 \cup E_1$ with respect to symmetry as before.

3. Implement a function that generates di-cographs randomly, e.g., by generating binary discriminating di-cotrees.

   Option A:

   - generate a random, strict, binary tree
   - label inner nodes as 0, 1 or $\overrightarrow{1}$
   - if there is an edge with same labels at its nodes, contract the edge
   - use function `cotree_to_rel` followed by `rel_to_fitch` to create the graph

   Option B:

   - define a nodeset
   - generate a random order of operations
   - build graph accordingly

4. Implement a function that creates partial tuples $\mathcal{E} = (E_0, E_1, E_{\overrightarrow{1}})$ by deleting $10, 20, \ldots,$ 90% of the information in the di-cographs.

5. Benchmark Algorithm 1 and Algorithm 2 on the partial datasets (Fitch and cograph based).

   - both algorithms are implemented already in the library as `algorithm_one` and `algorithm_two`
   - in Algorithm 2, choose $w(x :: y) = 0$ when $x :: y \in \overline{\mathcal{E}}$ (assign to $m_0$ your favorite positive number).

6. How many of the provided partial relations of random cographs are Fitch-sat?

   - use function `check_fitch_graph`

7. Compare the performance of Algorithm 1 and Algorithm 2 by measuring the difference between the recovered and the original Fitch graphs. The differences can be measured by the relative symmetric difference of the corresponding full tuples, see Section 5 for its definition.

### 3.3 WP2

The scope of this workpackage is to implement new heuristics for the weighted Fitch graph problem (on an empty partial tuple).

1. Familiarize yourself with the `generate_weights` function described in Section 5. For a given Fitch-graph, generate weights using (at least) two different distributions - one for edges/relations present in the Fitch-graph, and one for relations *not* present in the Fitch-graph.

2. Familiarize yourself with the top-down recursive algorithm framework provided below and with the corresponding function `partition_framework`. Either make sure that your partition algorithms are compatible with the framework provided, or implement your own framework realizing the algorithm below.

3. Let $(P_1, P_2)$ be a bipartition of the vertex-set of a digraph $G$. The edges cut is $\{(x, y) \in E(G) \,|\, x \in P_1 \text{ and } y \in P_2\}$. Implement a greedy heuristic to bipartition $V(G)$ using two different scoring functions (1) the average weight of the edges cut and (2) the sum of the weights of the edges cut. Take care that your scoring functions are compatible with the provided framework (or with your own, in case you chose to implement it yourself).

4. Implement a random partition function of $V(G)$.

5. Implement Louvain algorithm to bipartition $V(G)$ for average weight per edge cut (if Louvain algorithm does not give a bipartition, split the partition maximizing the average weight per edge cut).

6. Implement Leiden algorithm to bipartition $V(G)$ for average weight per edge cut (if Leiden algorithm does not give a bipartition, split the partition maximizing the average weight per edge cut). Fix $\lambda = 1$ at first. Later you may systematically vary $\lambda$ to benchmark its influence.

7. Benchmark your versions of the top-down approach on the set of weights generated in Task 1.

8. Compute the relative symmetric difference between the inferred and original Fitch graphs for each of the six methods.

9. Compare the performance of the methods (Hint: violin plots or box plots might be a good fit).

***Top-Down Approach.*** Let $G_0$, $G_1$, $G_{\overrightarrow{1}}$ be three (auxiliary) undirected complete and weighted graphs on $V(G)$ associated to the weights of the corresponding relation defined as follows; the edge weight of $(x, y)$ is $w(x \smile y)$ (resp. $w(x \rightleftharpoons y)$) in $G_0$ (resp. $G_1$), and $\frac{w(x \to y)+w(y \to x)}{2}$ in $G_{\overrightarrow{1}}$, as the relation $\overrightarrow{1}$ is (not necessarily) symmetric. The top-down approach consists of recursively partitioning the auxiliary graphs to build a Fitch-cotree. At each recursive step, we greedily choose the partition on one of the three graphs, and introduce a corresponding inner node in the Fitch-cotree; see Figure 5.

```
INPUT: vertex-set $V$, a set of weights $\mathcal{W}$, forbidden partitions $\mathcal{F} := \emptyset$, partition function
part, partition scoring function score.
OUTPUT: A Fitch-cotree $t$.

   PartitionHeuristic($\mathcal{W}, \mathcal{F}, V,$ part, score) :
     if $|V| = 1$ then return $((V, \emptyset), \emptyset)$        //unlabeled tree with a single node
     else CREATE  $G_0$, $G_1$, $G_{\overrightarrow{1}}$ from $\mathcal{W}$
          $G_0^l, G_0^r$ = part($G_0$)                  //part should return vertex sets
          $G_1^l, G_1^r$ = part($G_1$)
          $G_{\overrightarrow{1}}^l, G_{\overrightarrow{1}}^r$ = part($G_{\overrightarrow{1}}$)
          $s_0$ = score($G_0^l, G_0^r$)
          $s_1$ = score($G_1^l, G_1^r$)
          $s_{\overrightarrow{1}}$ = $max$(score($G_{\overrightarrow{1}}^l, G_{\overrightarrow{1}}^r$), score($G_{\overrightarrow{1}}^r, G_{\overrightarrow{1}}^l$))
          if $s_0 > s_1, s_{\overrightarrow{1}}$:
            $t_l$ = PartitionHeuristic($\mathcal{W}, \mathcal{F} \cup \{1, \overrightarrow{1}\}, G_0^l$, part, score) :
            $t_r$ = PartitionHeuristic($\mathcal{W}, \mathcal{F} \cup \{1, \overrightarrow{1}\}, G_0^r$, part, score) :
            return $t_l$ and $t_r$ joined under root with label 0.
          if $s_1 > s_0, s_{\overrightarrow{1}}$ and $1 \notin \mathcal{F}$:
            $t_l$ = PartitionHeuristic($\mathcal{W}, \mathcal{F}, G_1^l$, part, score)
            $t_r$ = PartitionHeuristic($\mathcal{W}, \mathcal{F}, G_1^r$, part, score)
            return $t_l$ and $t_r$ joined under root with label 1.
          if $s_{\overrightarrow{1}} > s_1, s_0$ and $\overrightarrow{1} \notin \mathcal{F}$:   //decide whether $G_{\overrightarrow{1}}^l \to G_{\overrightarrow{1}}^r$ or $G_{\overrightarrow{1}}^l \leftarrow G_{\overrightarrow{1}}^r$
            $t_l$ = PartitionHeuristic($\mathcal{W}, \mathcal{F} \cup \{1\}, G_{\overrightarrow{1}}^l$, part, score)
            $t_r$ = PartitionHeuristic($\mathcal{W}, \mathcal{F}, G_{\overrightarrow{1}}^r$, part, score)
            return $t_l$ and $t_r$ joined under root with label $\overrightarrow{1}$.      //order matters!
```

# 4 Dataset

Using the Asymmetree python package Schaller et al. [2022], we simulated 100 gene family histories (GFHs) on $n = 10, 15, 20, 25, 30$ leaves with an average of $n/3$ genes per species. As depicted in Figure 1, Asymmetree generates a species tree together with an embedded gene tree. In order to generate the gene trees we used different sets of parameters that correspond to evolutionary events in the gene trees. The evolutionary events considered are duplication

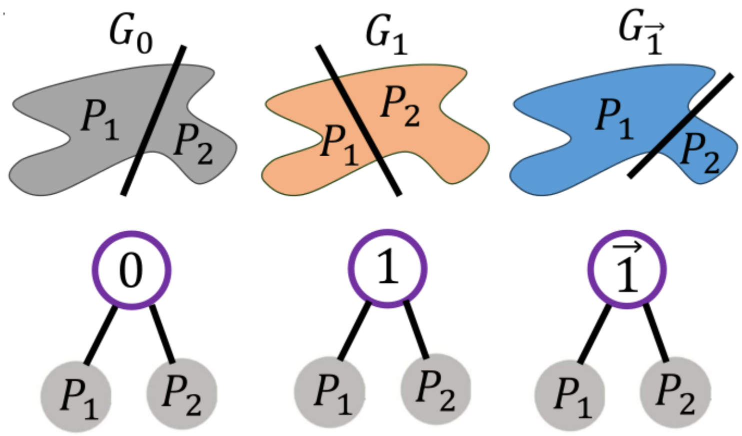

Figure 5: Schematic depiction of the graph partitioning of $G_0$, $G_1$, and $G_{\vec{1}}$. Each partition can be assigned to a score, for example, the sum of edge weights between $P_1$ and $P_2$ in the corresponding graphs. If we decide on a particular partition (of a particular $G_*$), this introduces an inner node in the Fitch-cotree with label $*$. Continuing to successively partition $P_1$ and $P_2$ in a way that avoids forbidden structures in Figure 2 leads to a-Fitch cotree.

(D), loss (L), and horizontal gene transfer (H), and we used seven different combinations of them for generating the dataset. We used the simulated GFHs to derive their corresponding xenology graphs, as in Figure 1. The dataset is structured as follows.

- **n10**
  - **D0.3_L0.3_H0.9** (Corresponding to different evolutionary rates)
    - **D0.3_L0.3_H0.9_n0**
      - **dFitch.graphml** (original xenology (Fitch) graph derived from the GFH)
      - **S_newick.json** (simulated species tree) with $n/3$ number of species
      - **T_newick.json** (simulated gene tree)
      - **biRelations.pkl**
        (all bidirectional relationships from dFitch.graphml as list of tuples) with $N$ number of genes
      - **uniRelations.pkl**
        (all unidirectional relationships from dFitch.graphml as list of tuples)
      - **emptyRelations.pkl**
        (all empty relationships from dFitch.graphml as list of tuples
    - **...**
    - **D0.3_L0.3_H0.9_n99** (Corresponding to different evolutionary rates)
  - **D0.5_L0.5_H0.5**
  - **...**
- **n15 ...**

- ◦ **...**
- ◦ **n30 ...**

The Fitch graphs are provided in `graphml` format which can be easily read by the `networkx.read_graphml` function. The different relationship files are serialized via the python built-in `pickle` library and can be accessed as

```
import pickle
with open("biRelations.pkl", "rb") as biRelations:
  myrelations = pickle.load(biRelations)
```

# 5 Useful functions

To allow for a smooth start, we decided to provide some basic functions that are tuned to this project. The functions are in the `lib.py` module that can be obtained by cloning the repository link in **WP0**. We strongly recommend you to use `python` as main programming language in this practical, as this will allow us supervisors to help in case there are problems with your code. Other programming languages can be used as well, but support from us strongly depends on whether we can actually use the chosen language in a meaningful way ourselves. If you desperately want to use a language different from python, make sure it comes with a graph library that implements basic graph theory algorithms (connected components, ...) as well as ideally some partitioning algorithms. You will also need functionality to read `.graphml` files as well as ideally reading of python serialized objects (`pickle`).

***Extracting Relationships*** ***`graph_to_rel(graph).`*** To allow to work with networkx graphs as well as sets, or lists, of relationships exchangeable we provide this function. Simply pass it a networkx DiGraph object to extract the corresponding unidirectional, bidirectional, or empty relationships from it. The functions returns a dictionary with keys 0 (no edge), 1 (bidirectional edge), and $d$ (unidirectional edge). For example, dictionary[1] contains all bidirectional edges.

***Relation to Fitch graph*** ***`rel_to_graph(unidrectional, bidirectional, empty).`*** Passing a dictionary of the corresponding relationships (as described above), the function creates and returns a networkx DiGraph object that represents the underlying Fitch graph.

***Check Fitch Graph*** ***`check_fitch_graph(graph).`*** Given a networkx DiGraph, the function returns `True` if provided graph is a Fitch graph, and `False` otherwise.

***Generate Weights*** ***`generate_weights(relation, distribution, parameters, symmetric=True)`*** . Given a set of relations, or list of edges, and a distribution function (for example, `numpy.random.normal`), the function `generate_weights` assigns at each relation a weight drawn from the provided distribution. Standard mode is `symmetric=True`, where the same weight is assigned to each symmetric pair (i.e. $(1,2)$ and $(2,1)$ will have the same weight). The parameters should be provided as list, and are passed to the chosen distribution function. For example, for (numpy.random.normal) the parameters `list(3, 0.75)` could be provided, and `generate_weights` would draw each weight from a normal distribution with mean 3 and standard deviation 0.75. `generate_weights` then returns a list of tuples where each tuple is structured as `(source,destination,weight)`.

***Symmetric Difference*** ***`sym_diff(graph, graph_hat)`.*** To compare two Fitch graphs with the same vertex-set $V$, we will use their (relative) symmetric difference, which counts the difference between edge-sets. The symmetric difference between Fitch graphs is defined in terms of their tuples. Let $\mathcal{E} = (E_0, E_1, E_{\overrightarrow{1}})$ be the tuple associated to the original Fitch graph and $\widehat{\mathcal{E}} = (\widehat{E_0}, \widehat{E_1}, \widehat{E_{\overrightarrow{1}}})$ the tuple of the inferred Fitch graph, the symmetric distance between them is $d_R(\mathcal{E}, \widehat{\mathcal{E}}) := d(\mathcal{E}, \widehat{\mathcal{E}})/|V|(|V|-1)$ where

$$d(\mathcal{E}, \widehat{\mathcal{E}}) = |(E_0 \cup \widehat{E}_0) \setminus (E_0 \cap \widehat{E}_0) \cup (E_1 \cup \widehat{E}_1) \setminus (E_1 \cap \widehat{E}_1) \cup (E_{\overrightarrow{1}} \cup \widehat{E}_{\overrightarrow{1}}) \setminus (E_{\overrightarrow{1}} \cap \widehat{E}_{\overrightarrow{1}})|.$$

Note that when the Fitch graphs coincide, $d_R(\mathcal{E}, \widehat{\mathcal{E}}) = 0$; on the other hand, when they do not share any relations, $d_R(\mathcal{E}, \widehat{\mathcal{E}}) = 1$.

***Algorithm 1*** ***`algorithm_one(relation, nodes, order)`*** . This function implements Algorithm 1 described above. Given the partial tuple of relations, the node set, and the order of rules applied, i.e. $(0, 1, 2)$ for $S1$ applied before $S2$, etc., the functions returns a full tuple in the shape described for function `graph_to_rel`.

***Cotree to Relation*** ***`cotree_to_rel(cotree)`*** . Given a cotree, this function returns the tuple (dictionary) of relations as described above.

***Algorithm 2*** ***`algorithm_two(uni, bi, empty, nodes, order)`*** . Given dictionaries of weighted relations, where each dictionary contains keys for all edges with their corresponding weight (e.g. $uni[(1,2)] : w$), the function implements Algorithm 2. It returns a Fitch-cotree which can be parsed into a full tuple (i.e. dictionary) using the function `cotree_to_rel`.

***Recursive Partition Framework*** ***`partition_framework(uni, bi, empty, nodes)`*** . This function provides a framework for the recursive partition approach described above. It is still missing some components - namely the partition function and the scoring function which need to be implemented and integrated by you.

# References

Orit Adato, Noga Ninyo, Uri Gophna, and Sagi Snir. Detecting horizontal gene transfer between closely related taxa. *PLoS computational biology*, 11(10):e1004408, 2015.

Brian J Arnold, I-Ting Huang, and William P Hanage. Horizontal gene transfer and adaptive evolution in bacteria. *Nature Reviews Microbiology*, 20(4):206–218, 2022.

GD Paul Clarke, Robert G Beiko, Mark A Ragan, and Robert L Charlebois. Inferring genome trees by using a filter to eliminate phylogenetically discordant sequences and a distance matrix based on mean normalized blastp scores. *Journal of bacteriology*, 184(8): 2072–2080, 2002.

Christophe Dessimoz, Daniel Margadant, and Gaston H Gonnet. Dlight–lateral gene transfer detection using pairwise evolutionary distances in a statistical framework. In *Annual International Conference on Research in Computational Molecular Biology*, pages 315–330. Springer, 2008.

Manuela Geiß, John Anders, Peter F. Stadler, Nicolas Wieseke, and Marc Hellmuth. Reconstructing gene trees from Fitch's xenology relation. *J. Math. Biol.*, 77:1459–1491, 2018a. doi: 10.1007/s00285-018-1260-8.

Manuela Geiß, John Anders, Peter F Stadler, Nicolas Wieseke, and Marc Hellmuth. Reconstructing gene trees from fitch's xenology relation. *Journal of mathematical biology*, 77: 1459–1491, 2018b.

Marc Hellmuth and Guillaume E. Scholz. Resolving prime modules: The structure of pseudocographs and galled-tree explainable graphs, 2023. arXiv:2211.16854.

Marc Hellmuth, Peter F Stadler, and Sandhya Thekkumpadan Puthiyaveedu. Fitch graph completion. *arXiv preprint arXiv:2306.06878*, 2023.

Jeffrey G Lawrence and Howard Ochman. Amelioration of bacterial genomes: rates of change and exchange. *Journal of molecular evolution*, 44:383–397, 1997.

Matt Ravenhall, Nives Škunca, Florent Lassalle, and Christophe Dessimoz. Inferring horizontal gene transfer. *PLoS computational biology*, 11(5):e1004095, 2015.

David Schaller, Marc Hellmuth, and Peter F Stadler. Asymmetree: a flexible python package for the simulation of complex gene family histories. *Software*, 1(3):276–298, 2022.

Gur Sevillya, Orit Adato, and Sagi Snir. Detecting horizontal gene transfer: a probabilistic approach. *BMC genomics*, 21(1):1–11, 2020.

Shannon M Soucy, Jinling Huang, and Johann Peter Gogarten. Horizontal gene transfer: building the web of life. *Nature Reviews Genetics*, 16(8):472–482, 2015.