Merging partially labelled trees: hardness and a declarative programming solution

Anthony Labarre, Sicco Verwer

Merging partially labelled trees: hardness and a declarative programming solution

Anthony Labarre Sicco Verwer

***Abstract*—Intraspecific studies often make use of haplotype networks instead of gene genealogies to represent the evolution of a set of genes. Cassens et al. [3] proposed one such network reconstruction method, based on the global maximum parsimony principle, which was later recast by the first author of the present work as the problem of finding a minimum common supergraph of a set of t partially labelled trees. Although algorithms have been proposed for solving that problem on two *graphs*, the complexity of the general problem on trees remains unknown. In this paper, we show that the corresponding decision problem is NP-complete for $t = 3$. We then propose a declarative programming approach to solving the problem to optimality in practice, as well as a heuristic approach, both based on the IDP system, and assess the performance of both methods on randomly generated data.**

***Index Terms*—Phylogenetic networks, supergraphs, NP-hardness, SAT solver, IDP.**

I. Introduction

Phylogenetic trees are the traditional tool for representing the evolution of a given set of species [6]. The last two decades, however, have witnessed the emergence of a new way of reconstructing and representing evolution, which has become widespread in phylogenetic studies: *phylogenetic networks*, which generalise phylogenetic trees by allowing multiple paths between species. The main reason for using networks rather than trees is that evolution is not always tree-like: genes may be duplicated, transferred or lost, and recombination events (i.e. the breaking of a DNA strand followed by its reinsertion into a different DNA molecule) as well as hybridisation events (i.e. the combination of genetic material from several species) are known to occur. Moreover, even when evolution is tree-like, situations exist in which a relatively large number of tree topologies might be "equally good", and not enough information is available to discriminate between those trees. One proposed solution to the latter issue is the use of *consensus trees*, where the idea is to find a tree that represents a compromise between the given topologies; another approach, on which we focus in this paper, is to build a *network* [7, 9] that is compatible with all topologies of interest.

Haplotype networks are used in the context of *intraspecific* studies, which focus on relations between genes rather than between species. Cassens et al. [3] proposed a new method for reconstructing such networks, based on a given set of trees rather than on the input sequences. Note that the trees studied in that context, namely, *gene genealogies*, differ from the typical phylogenetic trees studied in comparative genomics: whereas phylogenetic trees are usually binary (i.e. internal nodes have degree three), have labels attached only to their leaves, and contain branches of arbitrary real length, gene genealogies allow internal nodes of arbitrary degree, as well as labelled nodes that are not leaves, and their branches have length exactly one. Cassens et al.'s approach comprises two steps: *most parsimonious trees* are built from the sequences, and a subset of these trees is then merged into a graph. Their approach, which they refer to as "union of most parsimonious trees" (UMP), does not aim at building a smallest graph that contains *all* most parsimonious trees, as Bandelt et al. [1] did using *median networks*, but rather to summarise the information contained in a selected portion of those most parsimonious trees in a graph that is as "succinct" as possible.

The results produced by UMP on simulated data have been promising, compared with earlier algorithms [3]. However, the algorithm and the overall approach proposed by the authors lacked rigorous formalisation, and were later recast by the first author of the present work as a *minimum common supergraph problem*: given a set of partially labelled trees on the same label set, find a graph on the same vertex set which contains all input trees as subgraphs and which has as few edges as possible [10]. That work also contains two exact algorithms for the same problem on two partially labelled *graphs*, running in polynomial time under some assumptions and exponential time in the general case. To the best of our knowledge, the complexity of the problem has since remained open.

In this work, we settle the complexity of the above optimisation problem, by showing that the associated decision problem is NP-complete for three trees. We make up for this bad news by proposing a practical approach to solving the problem to optimality in practice, using the IDP system [13]. This allows us to model our minimum common supergraph problem as a constraint satisfaction problem that is automatically translated into a SAT instance and then solved quickly by a SAT solver. We give an exact and a greedy method for UMP, both based on this declarative programming approach, and assess the performances of both approaches on random instances of various sizes.

II. Background

We recall here a few definitions and notation that will be needed in the study of our problem, formally stated at the end of this section. Any graph-theoretical concept the reader might lack familiarity with can be found in any textbook on the topic, e.g. Diestel [5].

Definition II.1. [10] A *labelling* $\mathcal{L}$ for a subset U of vertices of a graph $G = (V, E)$ assigns a distinct label to each vertex in U; it is *partial* (resp. *complete*) if $U \subset V$ (resp. $U = V$), in which case we say that G is *partially* (resp. *completely*) *labelled*.

Unless explicitly stated, the label set will always be $\{1, 2, \ldots, k\}$, with $k \leq |V|$.

Definition II.2. [10] An (n, k)-*graph* $G = (V, E, \mathcal{L})$ is a graph on n vertices, k of which are labelled by $\mathcal{L}$.

Definition II.3. [10] An (n, k)-*tree* is a connected (n, k)-graph with $n - 1$ edges and whose labelled vertex set includes all vertices of degree 1.

The following function, which (possibly) returns the label of vertex v in the (n, k)-graph G, allows us to adapt classical graph-theoretical concepts to our needs:

$$lab : V(G) \to \{1, 2, \ldots, k\} \cup \{\varnothing\}$$
$$: v \mapsto lab(v) = \begin{cases} i & \text{if } v \text{ has label } i, \\ \varnothing & \text{otherwise.} \end{cases}$$

This is not to be confused with the labellings introduced in Definitions II.1 and II.2: labelling $\mathcal{L}$ *assigns* labels to vertices, while function lab (possibly) *returns* labels. We will also use lab on edges, in order to obtain the pairs of labels that correspond to the endpoints of interest: if v, $w \in V(G)$, then $lab(\{v, w\}) = \{lab(v), lab(w)\}$. Therefore, we have:

$$lab(E(G)) = \{\{i, j\} \mid i, j \in \{1, 2, \ldots, k\} \cup \{\varnothing\} \text{ and } \exists\, \{v, w\} \in E(G) : lab(v) = i, lab(w) = j\}.$$

Definition II.4. An (n, k)-graph G is a *subgraph* of an (n, k)-graph H if the labellings of G and H can be completed in such a way that the resulting (n, n)-graphs G' and H' satisfy $lab(E(G')) \subseteq lab(E(H'))$. In that case, we also say that H is a *supergraph* of G.

"Completing a labelling" means assigning distinct labels to the remaining unlabelled vertices; already labelled vertices must not be altered. In the following, the primed notation G' will always refer to a completely labelled graph obtained from an (n, k)-graph G by completing its labelling.

Definition II.5. [10] An (n, k)-graph G is a *common supergraph* of a set $\mathscr{G}$ of (n, k)-graphs if it is a supergraph of each element of $\mathscr{G}$. It is *minimum* if there is no other common supergraph of $\mathscr{G}$ with fewer edges.

Figure 1 shows two (n, k)-trees T_1 and T_2 along with two supergraphs G_1 and G_2 of $\{T_1, T_2\}$. Both G_1 and G_2 are minimal in the sense that deleting any of their edges invalidates the supergraph property, but only G_1 is minimum. Note that $\mathcal{L}(T_1)$ and $\mathcal{L}(T_2)$ cannot be completed in such a way that $lab(E(T_1')) = lab(E(T_2'))$ (again, T_1' and T_2' are (n, n)-trees obtained from T_1 and T_2 by completing those (n, k)-trees' labellings).

We now have everything we need to formally state our problem as a decision problem:

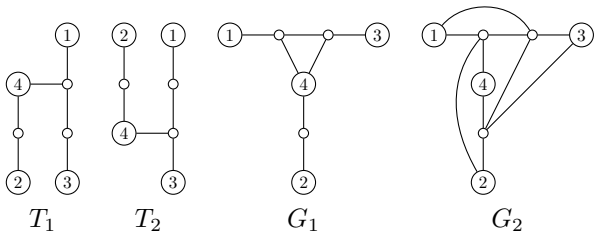

Fig. 1. Two $(7, 4)$-trees T_1 and T_2, and common supergraphs G_1 and G_2 of T_1 and T_2; G_1 is minimum, but G_2 is not.

COMMON SUPERGRAPH OF PARTIALLY LABELLED TREES (CS-PLT)

- **Instance:** (n, k)-trees T_1, T_2, …, T_t on the same label set, a natural upper bound K.
- **Question:** can the labellings of T_1, T_2, …, T_t be completed in such a way that $\cup_{i=1}^{t} lab(E(T_i')) \leq K$?

Note that a common supergraph of the input trees is defined exactly by the above union.

III. The complexity of CS-PLT

In this section, we prove the hardness of CS-PLT.

Theorem III.1. CS-PLT *is* NP*-complete for three trees.*

Proof: We present a reduction from MONOTONE 1-IN-3 SATISFIABILITY (see Schaefer [12]):

MONOTONE 1-IN-3 SATISFIABILITY

- **Instance:** a Boolean formula $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ without negations over a set $\Sigma = \{\ell^1, \ell^2, \ldots, \ell^n\}$, with exactly three distinct literals per clause.
- **Question:** does there exist an assignment of truth values $f : \Sigma \to \{\text{TRUE}, \text{FALSE}\}$ such that exactly one literal is TRUE in every clause of ϕ?

a) The transformation: We encode instances of MONOTONE 1-IN-3 SATISFIABILITY using three trees, whose construction and purpose are explained below, and we illustrate the transformation on an example in Figure 2.

1) The first tree T_1 encodes the occurrences of literals in the MONOTONE 1-IN-3 SATISFIABILITY instance ϕ. It is constructed using a matrix indexed by the literals and clauses from ϕ. Every occurrence of a literal ℓ^j in a clause C_i is mapped onto a pair of nodes connected by an edge, where one node is a leaf labelled with L_i^j, which we call a *literal node*, and the other node is unlabelled. After creating these nodes for all literal occurrences, we connect the unlabelled nodes that are connected by an edge with occurrences of the same literal by adding edges vertically in the matrix, i.e., in order of occurrence. The first occurrence of every literal is then connected to a root node R, which is itself connected to a TRUE node T and a FALSE node F (all three nodes are labelled).
2) In tree T_2, R is connected to three paths:

a) a first path that consists of all $3m$ literal nodes in an arbitrary order (without loss of generality);
b) a second path, called the TRUE CHAIN, which contains m unlabelled nodes and ends with T;
c) a third path, called the FALSE CHAIN, which contains $2m$ unlabelled nodes and ends with F.

The first path is connected to node R, while the unlabelled extremities of the TRUE CHAIN and of the FALSE CHAIN are both connected to R. The TRUE CHAIN and the FALSE CHAIN represent a truth assignment to the literals in ϕ. This assignment is determined by labelling T_1 and T_2 in the CS-PLT instance: a literal ℓ^j in C_i represented by edge $\{L_i^j, u\}$ in T_1 is set to TRUE (resp. FALSE) if u is assigned the same label as a node from the TRUE CHAIN (resp. FALSE CHAIN) from T_2.

3) Tree T_3 overlaps for a large part with T_2. The only difference is that the TRUE CHAIN is split up and every unlabelled node from this chain is connected to T and three literal nodes from a unique clause. These edges thus encode the different clauses in ϕ. In addition, by limiting the number of allowed edges in a CS-PLT solution by a value K, they encode the constraint that every clause contains exactly one TRUE literal. Note that a minimal CS-PLT solution assigns the same labels to the TRUE chains from T_2 and T_3, and the same labels and label ordering to the FALSE chains.

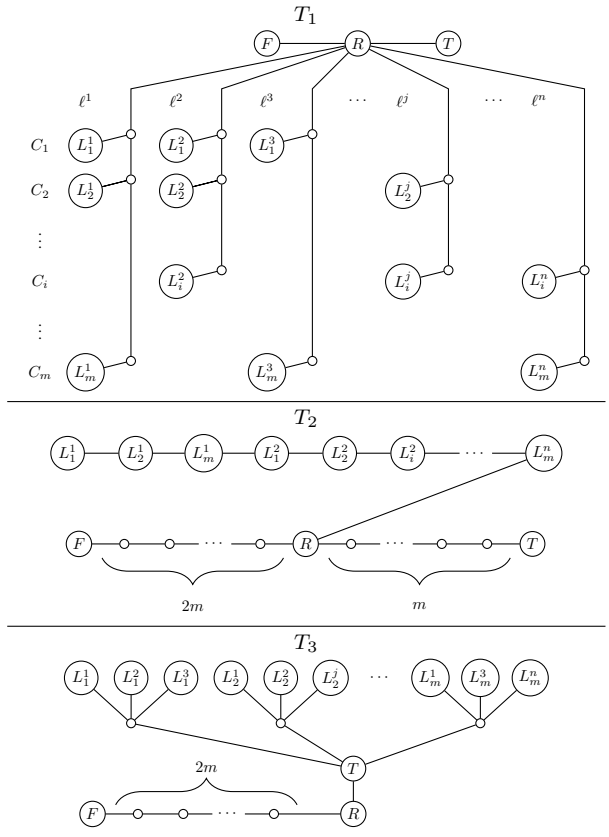

Fig. 2. The three trees built in our transformation.

Figure 3 page 5 shows an example of the construction applied to a small example instance (ignore labels 1 to 12 for now). In addition to these trees, the CS-PLT decision problem requires an upper bound K, which we derive later in the proof:

$$K = 12m + n + 1.$$

We now show that ϕ is satisfiable under the monotone 1-in-3 restrictions if and only if the labellings of these three trees can be completed in such a way that the union of the resulting labelled edge sets has size at most K.

($\Rightarrow$): Let f be a solution to ϕ. We use f to construct a solution to the CS-PLT instance of size at most K, which consists of three respective labellings for the unlabelled nodes of T_1, T_2 and T_3, as follows.

1) We examine each path following the lexicographical order on literals, and follow paths downwards from R, assigning and incrementing labels as we go, starting with 1. More formally, every unlabelled node U_i^j connected to a literal node L_i^j in T_1 receives the label $a(U_i^j)$ defined below and which corresponds to the number of literal nodes $L_{i'}^{j'}$ connected to unlabelled nodes $U_{i'}^{j'}$ representing either literals with lexicographically smaller labels alphabetically (i.e. $\ell^{j'} < \ell^j$) or the same literal but occurring in an earlier clause (i.e. $\ell^{j'} = \ell^j$ and $i' < i$):

$$a(U_i^j) = |\{L_{i'}^{j'} \mid (\ell^{j'} < \ell^j) \vee (\ell^{j'} = \ell^j \wedge i' < i)\}|.$$

2) The k^{th} unlabelled node from the TRUE CHAIN $U_{k,T}$ in T_2 (ordered from R to T) receives the label assigned to the k^{th} unlabelled node U_i^j in T_1 (in ascending label order) that represents a TRUE literal:

$$\begin{aligned}&(a(U_{1,T}), \ldots, a(U_{m,T}))\\ &= \text{SORT}(\{a(U_i^j) \text{ such that } f(\ell^j) = \text{TRUE}\}).\end{aligned}$$

Since f is a 1-in-3 solution, it is guaranteed that this assigns a unique label to every unlabelled node from the TRUE CHAIN.

3) Similarly, the i^{th} node from the FALSE CHAIN in T_2 and T_3, namely, $U_{i,F}$, receives the label of the U_i^j nodes representing FALSE literals:

$$\begin{aligned}&(a(U_{1,F}), \ldots, a(U_{2m,F}))\\ &= \text{SORT}(\{a(U_i^j) \text{ such that } f(\ell^j) = \text{FALSE}\}).\end{aligned}$$

4) The i^{th} split up TRUE CHAIN nodes $U_{k,s}$ from T_3 are all connected to all three literal nodes from clause C_k. We label these nodes with the label of the U_i^j node representing the TRUE literal ℓ^j in C_i:

$$a(U_{i,s}) = a(U_i^j) \text{ such that } f(\ell^j) = \text{TRUE}.$$

The labellings are uniquely defined since f assigns the TRUE value to exactly one literal in every clause C_i. Figure 3 shows the completely labelled trees that result from applying the aforementioned steps to the trees constructed from an example instance of MONOTONE 1-IN-3 SATISFIABILITY.

We now show that these labellings yield a graph that contains exactly $K = 12m + n + 1$ edges. Every tree potentially adds all its $6m+2$ edges to the resulting graph, so we derive K by counting the overlapping edges between the different trees. Following the definition of CS-PLT, the completely labelled trees we obtained will be denoted by T_1', T_2' and T_3'. Let us add all edges from T_1' and T_3' to T_2'; we make the following observations:

- The $2m+1$ edges connecting the FALSE CHAIN nodes to R in T_3' already appear in T_2', since the unlabelled nodes are assigned exactly the same label by $a(\cdot)$. Moreover, exactly one of the edges between T and the split up TRUE CHAIN in T_3' appears in the TRUE CHAIN in T_2';
- Tree T_1' contains a lot of edges already in T_2' or T_3' due to the $a(\cdot)$ labelling:
  - m of the edges to literal nodes overlap with those from T_3' because every $U_{k,s}$ is assigned the same label as some U_k^j.
  - All of the $3m-n$ edges connecting the unlabelled nodes U_i^j connect nodes representing literals that are assigned the same truth value by f and consecutive labels by a, so these nodes are already connected either by the TRUE CHAIN or by the FALSE CHAIN in T_2'.
  - 2 edges between R and newly labelled nodes overlap with those in T_2 since the TRUE CHAIN and FALSE CHAIN start with the smallest label assigned to a TRUE and a FALSE literal by a, corresponding to the first occurrence of these literals.
  - 1 edge connecting R with T.

This sums up to $3(6m + 2) - (2m + 1 + 1) - (m + 3m - n + 2 + 1) = 12m + n + 1$ edges, which equals K.

($\Leftarrow$)*:* If the constructed CS-PLT instance is true, then the original MONOTONE 1-IN-3 SATISFIABILITY instance is true. We first observe that the value we derived for K in the ($\Rightarrow$) part is the minimum number of edges that can be obtained by labelling our three trees and taking the union of the resulting edge sets since we counted the maximum number of overlapping edges, making in total:

- 2 edges due to T_1 ($\{R, T\}$ and $\{R, F\}$);
- n edges connecting R with unlabelled nodes, which is minimal due to T_1;
- $5m$ edges between literal nodes and unlabelled nodes, minimal due to T_1 and T_3;
- $3m$ edges connecting literal nodes, minimal due to T_2;
- $2m - 1$ edges between unlabelled nodes in the FALSE CHAIN, minimal due to T_2;
- $m - 1$ edges between unlabelled nodes in the TRUE CHAIN, minimal due to T_2;
- m edges between T and unlabelled nodes in the split up TRUE CHAIN, minimal due to T_3;
- and 1 edge between F and an unlabelled node in the FALSE CHAIN, minimal due to T_2.

This sums up to $12m + n + 1 = K$.

We note that after completing the labelling of T_1, T_2 and T_3, we necessarily obtain a common supergraph $G = (V, E)$ such that:

$$1 \leq |\{\{v, w\} \in E : lab(v) \in L_i^j\}| \leq 2.$$

In other words, every literal node is adjacent to at least one and at most two newly labelled nodes, independent of the labelling. Since we counted exactly $5m$ edges between these nodes, and there are $3m$ literal nodes, this gives exactly m literal nodes that are connected by a single edge with a newly labelled node. Consequently, for every split up TRUE CHAIN node from T_3, exactly one node receives a label such that one of its three edges with literal nodes overlaps with an edge from T_1. The literal nodes connected to these overlapping edges determine the TRUE literals in the original satisfiability problem, all other literals are set to FALSE. Since every split up TRUE CHAIN node in T_3 is connected to nodes representing exactly the literal occurrences of a single clause, this makes exactly one literal TRUE in every clause.

In addition to this property, we require that if a literal is true, then every instance of that literal is TRUE. We show this by making the following observation that is key to our translation:

> A CS-PLT solution of size K has no edges between the TRUE CHAIN of T_2 and the FALSE CHAIN of T_3.

To see why this holds, one only has to observe that in the above edge counts, we counted exactly $3m - 2$ edges between unlabelled nodes. Since the TRUE CHAIN and the FALSE CHAIN in T_2 already contribute this amount of edges, any additional edge will yield a solution of size $K + 1$. Thus, in a solution of size K, the same labels are assigned to the FALSE CHAIN nodes from T_2 and T_3. Furthermore, all of the edges between unlabelled nodes from T_1 have to overlap with those from T_2. Since these edges connect the different occurrences of literals, these occurrences are all labelled with either TRUE CHAIN or FALSE CHAIN labels, but not both. Consequently, if the CS-PLT problem is true (has a solution of size K), then using our construction, every literal occurrence of the same literal is assigned the same truth value, and exactly one literal is set to TRUE in every clause, making the MONOTONE 1-IN-3 SATISFIABILITY instance satisfied.

b) Time complexity: The transformation clearly runs in time polynomial in the size of the MONOTONE 1-IN-3 SATISFIABILITY instance, and a solution to CS-PLT can easily be verified in polynomial time. The CS-PLT problem is therefore NP-complete. ■

IV. Finding a minimum common supergraph in practice

The hardness of CS-PLT motivates our search for efficient exact or approximate solutions. In that spirit, we decided to *translate* our problem into a constraint satisfaction problem, and to rely on an efficient SAT *solver* to obtain an exact solution to it.

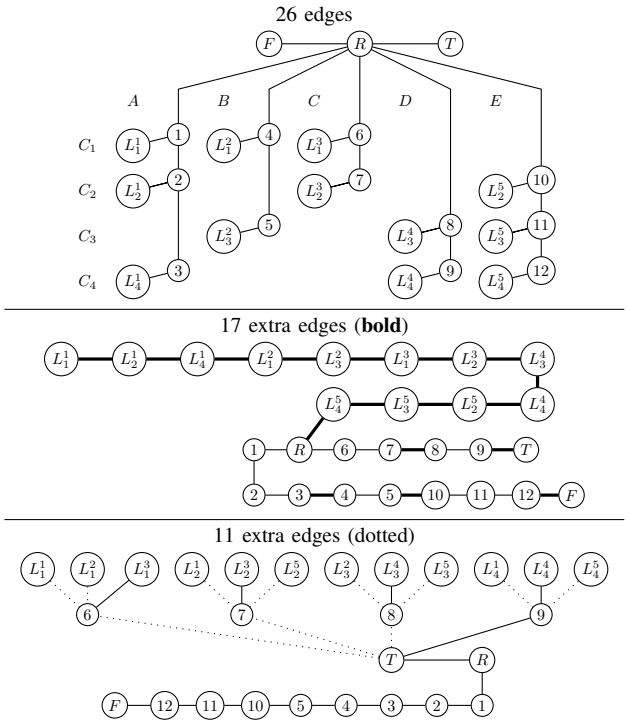

Fig. 3. A solution to the CS-PLT instance constructed from the MONOTONE 1-IN-3 SATISFIABILITY instance $(A \vee B \vee C) \wedge (A \vee C \vee E) \wedge (B \vee D \vee E) \wedge (A \vee D \vee E)$, which has as satisfying assignment $f(C) = f(D) =$ TRUE. The union of the labelled edge sets has size $26 + 17 + 11 = 54 = 12m + n + 1$, with $m = 4$ and $n = 5$.

Figure 4 shows the typical workflow of a SAT solver based approach. We circumvent the difficulties pointed out in that workflow by relying on the IDP model expansion system [13], which merely requires us to provide a logical description of our problem and a specific instance. IDP translates the description into a constraint satisfaction problem, runs a solver, and translates the result back into a solution to our problem. Another attractive feature of IDP is that it can be used as an *anytime algorithm*: one can terminate the solving process before its completion and retrieve the best solution found so far.

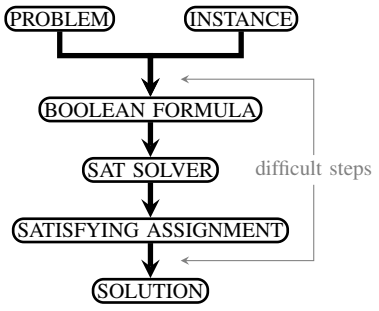

Fig. 4. The typical workflow of a SAT solver based approach.

We give an exact approach (Section IV-D) and a greedy approximation (Section IV-E) in the following sections, starting with an introduction to SAT solvers in Section IV-A. We then describe IDP, its input and two models in Sections IV-B to IV-D, and explore their efficiency in practice on artificial data in Section IV-F.

A. Satisfiability and SAT solvers

The NP-complete *satisfiability problem*, which we recall below for completeness, is central to the field of computational complexity theory [4].

SATISFIABILITY (SAT)
- **Instance:** a Boolean formula ϕ in conjunctive normal form.
- **Question:** is there a satisfying assignment for ϕ?

SAT and its variants have spawned tremendous interest among researchers, who have developed a number of practical and efficient algorithms, generally referred to as SAT *solvers*, for solving instances of those problems in practice (see e.g. Gomes et al. [8] for a recent account). A number of highly-optimised implementations exist, which makes it possible to solve several well-known hard problems to optimality in a reasonable amount of time in many cases. One of the difficulties lies in formulating the problem as a satisfiability problem [2, ch. 2]; fortunately, the IDP system, described below, makes this step a lot easier.

B. The IDP system

The IDP system [13] consists of two parts: a grounder [14] and a solver [11]. The grounder (GIDL) transforms a search or optimisation problem into a propositional formula that can be solved using the solver; the solver (MINISATID) then produces a solution if one exists. This provides an easy method for declarative problem solving: all we have to do is provide a high-level specification of our problem and of the instance we want to solve; the IDP system then determines, using searches and heuristics, a good formulation of this problem in propositional logic (i.e. as an efficiently solvable instance of SAT), and finally runs the solver, translating upon completion any solution it finds back to the high-level specification.

The IDP language is straightforward and easy to use, thanks to a multitude of logical operators, the ability to perform arithmetic operations, and the possibility of providing inductive definitions. The latter two in particular make it possible to define complex constraints or optimisation parameters in a neat and succinct way. Although such definitions would normally result in a blow-up of the propositional specification of the problem, the IDP solver contains specialised propagation mechanisms suitable for reasoning directly on such inductive definitions. These mechanisms are built on top of the popular MINISAT solver without sacrificing much performance. The ability to write complex problem descriptions in just a few lines of code makes it an ideal tool for testing different problem specifications, and is the main strength of the IDP system.

C. A basic model

Figure 5 shows an IDP model we designed to represent the optimisation version of CS-PLT. This model is basic,

but we show it nonetheless for clarity, and will improve it in Section IV-D. It consists of four sections:

1) the "Given:" section specifies the format of an instance (in our case, a list of edges for each tree, along with some labels that are already assigned to a few vertices in each tree);
2) the "Find:" section describes the format of a solution (in our case, a set of labelled edges);
3) the "Satisfying:" section specifies the constraints edges and labels are subject to; and finally,
4) the "Minimize:" section describes the function that a solution should optimise (in our case, the size of the union of the completely labelled edge sets).

```
Given:
  type int Tree
  type int Node
  type int Label
  partial PreLabel(Tree, Node) : Label // some nodes are already labelled
  TEdge(Tree, Node, Node)

Find:
  Label(Tree, Node) : Label          // label the remaining nodes in each tree
  Edges(Label,Label)

Satisfying:
  { Edges(n,m) <- TEdge(t,x,y)          // once labelled, edges are assembled
                & Label(t,x) = n          // to build the common supergraph
                & Label(t,y) = m.
    Edges(n,m) <- Edges(m,n).                          // edges are undirected
  }

  ! t n : Label(t,n) = PreLabel(t,n).    // extant labels must not be changed
  ! t c : ?1 n : Label(t,n) = c. // use each label exactly once in each tree

Minimize:
  #{ x[Label] y[Label] : Edges(x,y) }          // the size of the supergraph
```

Fig. 5. The code used by the IDP system to model the optimisation version of CS-PLT.

Specifying an instance of CS-PLT in this format is easy. Figure 6 shows an example of a valid input, which consists of the following parts:

1) the `Tree` line specifies the unique indices from $\{1, 2, \ldots, t\}$ summarising our input (n, k)-trees T_1, T_2, ..., T_t;
2) the `Node` and `Label` lines specify the set $\{1, 2, \ldots, n\}$ of indices and labels used to refer to vertices;
3) the `PreLabel` set specifies the labellings $\mathcal{L}_1$, $\mathcal{L}_2$, ..., $\mathcal{L}_t$, where `i, v -> b` means that vertex `v` in tree T_i has label `b`; and
4) the `TEdge` section specifies each tree's edge set, where `i, u, v` means that $\{u, v\} \in E(T_i)$.

```
Tree = { 1; 2; 3; 4; 5 }                                  // ID's used for the trees
Node = { 1 .. 8 }                                        // ID's used for the vertices
Label = { 1 .. 8 }                                        // the range used for labels
PreLabel = {                                    // the labelled nodes in each tree
        1, 1->4; 1, 5->2; 1, 7->1; 1, 8->3;
        2, 1->1; 2, 5->2; 2, 6->3; 2, 8->4;
        3, 1->4; 3, 2->2; 3, 4->1; 3, 8->3;
        4, 1->4; 4, 3->2; 4, 4->1; 4, 8->3;
        5, 1->3; 5, 3->1; 5, 7->2; 5, 8->4;
}
TEdge = {                                       // the set of edges in each tree
        1, 1, 3; 1, 6, 7; 1, 2, 8; 1, 1, 4; 1, 1, 6; 1, 2, 4; 1, 3, 5;
        2, 1, 2; 2, 4, 6; 2, 4, 8; 2, 5, 7; 2, 2, 3; 2, 3, 7; 2, 2, 4;
        3, 4, 7; 3, 6, 7; 3, 5, 7; 3, 3, 8; 3, 1, 5; 3, 3, 6; 3, 2, 5;
        4, 1, 2; 4, 4, 7; 4, 5, 6; 4, 5, 7; 4, 3, 6; 4, 2, 5; 4, 7, 8;
        5, 2, 7; 5, 2, 6; 5, 4, 8; 5, 4, 5; 5, 1, 5; 5, 3, 6; 5, 2, 5;
}
```

Fig. 6. An example of an instance of our problem formatted for use by the IDP system; in this case, the instance consists of five $(8, 4)$-trees.

Given this input, IDP will try to find an assignment to `Edges` and `Label`, specified in the `Find:` part, that satisfies all constraints specified in the `Satisfying:` part. Once a solution has been found, IDP records the number of `Edges`, specified in the `Minimize:` part, and automatically adds clauses that force the underlying solver to try to find another solution with fewer edges. This continues until the solver is unable to find new solutions, or proves that the remaining problem is unsatisfiable. The last solution (assignment to `Edges` and `Label`) is returned by IDP, and its edges constitute a (minimum) common supergraph of the input trees.

D. An improved model

The model described in Section IV-C lacks efficiency. We identify two reasons for this lack of speed: differently labelled solutions can yield isomorphic supergraphs, and the definition of edges produces an unnecessarily difficult SAT instance. We address these issues by adding *symmetry breaking* predicates, and by defining a completely labelled edge set for each tree instead of a "global" supergraph edge set.

a) Symmetry breaking: Labellings merely match vertices in different trees; the actual labels do not matter, and permuting the labels assigned to the initially unlabelled vertices in any tree will not affect the size of the solution if we permute the corresponding labels in the other trees accordingly. Therefore, we can safely choose an arbitrary labelling for the unlabelled vertices of any one tree in our instance, thereby reducing the search space by a factor of $(n - k)!$.

b) Supergraph edges per tree: The way edges are defined in the model of Figure 5 results in an instance that is difficult to solve, which makes the model inefficient. The reasons why a particular model is inefficient are unfortunately not always obvious; models that yield SAT instances with fewer clauses are usually regarded as more efficient, but sometimes larger models and redundant clauses have a positive effect on the runtime of a SAT solver [2]. We identified by trial-and-error three inefficiencies in the definition of edges in the model of Figure 5, which we list and address below.

1) A first cause of inefficiency is the way in which the edges of the supergraph are specified as being undirected. In Figure 5, this is specified using the labels of nodes, and in an inductive way. Since these labels are free variables, and the nodes in a tree are fixed by the model input, it is more efficient to specify this property using these nodes instead of their labels. We do so by adding an additional declaration for undirected edges:

   `UEdge(i,u,v)`, which is TRUE if and only if `TEdge(i,u,v)` or `TEdge(i,v,u)` is TRUE.

   Constraints are then specified using the `UEdge` variables instead of the `TEdge` variables.
2) A second cause of inefficiency that we discovered is related to the way in which MINISATID makes use of

the clauses. For reasons that remain to be investigated (likely due to propagation mechanisms), MINISATID is able to find satisfying assignments much more quickly when the edges of the common supergraph are specified per tree:

`TreeEdge(i,n,m)`, which is TRUE if `UEdge(i,u,v)` is TRUE and `Label(i,v)=n` and `Label(i,u)=m`.

An element of `Edges` is then TRUE if and only if there exists a corresponding `TreeEdge`.

3) A third and final cause of inefficiency is already visible in the `TreeEdge` definition. Instead of an equivalence constraint (if and only if), we require an implication (if). This means that a `TreeEdge(i,n,m)` can be TRUE even though the nodes with labels `n` and `m` are not joined by an edge in tree `i`. However, since the aim is to minimise the number of `Edges` and therefore the number of `TreeEdges`, this constraint is implicit in the model. Requiring `TreeEdges` (or `Edges`) to be FALSE when there is no corresponding edge in a tree is redundant information. In our experience, removing this information results in an improved performance of MINISATID.

Figure 7 shows the improved model that we used in the experiments.

```
Given:
  type int Tree
  type int Node
  type int Label
  partial PreLabel(Tree, Node) : Label     // some nodes are already labelled
  TEdge(Tree, Node, Node)

Find:
  UEdge(Tree,Node,Node)
  TreeEdges(Tree,Label,Label)
  Edges(Label,Label)

Satisfying:
  { UEdge(t,x,y) <- TEdge(t,x,y) | TEdge(t,y,x). }        // undirected edges

  ! t x y n m : ( n < m &                                  // restrict edge values
                  Label(t,x) = n &                         // in every tree
                  Label(t,y) = m ) =>       // the colours of connected nodes
                (UEdge(t,x,y) => TreeEdges(t,n,m)).       // share a tree edge

  ! t n m : n < m =>                                     // assemble tree edges into
            (TreeEdges(t,n,m) => Edges(n,m)).           // the common supergraph

  ! n m : n >= m => ~Edges(n,m).          // fix values of unneeded variables
  ! t x : Label(t,x) = PreLabel(t,x).    // extant labels must not be changed
  ! t n : ?1 x : Label(t,x) = n. // use each label exactly once in each tree

Minimize:
  #{ n[Label] m[Label] : Edges(n,m) }          // the size of the supergraph
```

Fig. 7. An improvement over the model shown in Figure 5.

E. The greedy approach

We implemented the following greedy approach in addition to our exact approach:

1) find a minimum common supergraph for every pair of trees using IDP;
2) merge the two trees that yield the smallest common supergraph G, and replace them with G;
3) for every remaining tree T, use IDP to compute a minimum common supergraph of T and G;
4) merge G with the tree T that adds the fewest edges to G, and replace G and T with their resulting supergraph H;
5) go back to step 3 if any tree remains.

Merging one tree at a time with the current common supergraph greatly reduces the search space. The idea of carrying out the merging process in a way that minimises the number of edges added at each step seems sensible, but it is not necessarily optimal, as Figure 8 shows. An interesting open question is whether the ratio between the solution found using an optimal pairwise merging strategy and the optimal solution is bounded. In our experiments (Section IV-F), the greedy method performed very well, significantly outperforming the exact approach on larger problem instances where the solver timed out before reaching an optimal solution.

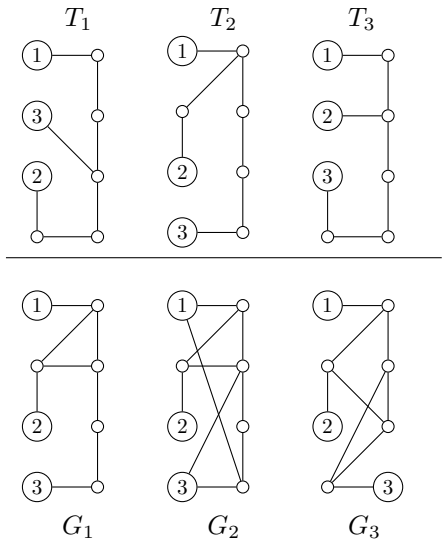

Fig. 8. An instance on which the greedy approach performs suboptimally. The first step creates a minimum common supergraph G_1 of $\{T_2, T_3\}$ with only one additional edge, then creates a minimum common supergraph G_2 of $\{G_1, T_1\}$ with 10 edges. However, G_3 is a common supergraph of $\{T_1, T_2, T_3\}$ with only 9 edges.

F. Experimental results

For our experiments[1], we generated random CS-PLT instances of varying difficulty. We generated four different instances for every setting of the following parameters: 5, 10, or 20 trees; 10, 20, or 50 nodes per tree; and 5, 10, or 25 labelled nodes per tree. Unlabelled trees are generated by randomly adding edges between a growing connected component and an isolated vertex; since the number of leaves in the resulting tree may exceed the number of labels, we then modify it by repeatedly connecting random pairs of leaves, and removing the existing edge incident to either leaf to avoid creating cycles. When we have enough labels, we then randomly assign them first to the leaves and then to the internal nodes.

We ran both the exact method and the greedy method on every generated instance. The exact method was given a maximum runtime of 2 000 seconds. Since even pairwise

[1]Experiments run on a desktop machine equipped with an Intel(R) Core TM i7 CPU 870 2.93GHz CPU (64bits) with 8GB of RAM.

mergers can take a long time, the greedy method was given at most 10 seconds for every pairwise merger. Table I reports on the average sizes per parameter setting of the solutions found by both methods.

			solution sizes	
#trees	#nodes	#labels	exact	greedy
5	10	5	17.50	18.00
10	10	5	19.50	21.50
20	10	5	23.00	25.25
5	20	5	34.75	**32.50**
5	20	10	53.00	**46.00**
10	20	5	38.75	**35.25**
10	20	10	64.25	**56.50**
20	20	5	42.25	42.25
20	20	10	75.50	**71.75**
5	50	5	130.00	131.25
5	50	10	128.00	132.75
5	50	25	207.75	**184.75**
10	50	5	183.75	**154.50**
10	50	10	177.75	**154.75**
10	50	25	270.00	**269.25**
20	50	5	241.50	**171.75**
20	50	10	232.00	**152.25**
20	50	25	346.25	**279.00**

TABLE I
AVERAGE SOLUTION SIZES OBTAINED BY THE EXACT AND THE GREEDY METHODS ON RANDOM INSTANCES WITH VARIOUS PARAMETERS AND PRESCRIBED TIMEOUTS. THE GREEDY APPROACH WAS ABLE IN SOME CASES (SHOWN IN **BOLD**) TO OUTPERFORM THE EXACT APPROACH.

IDP was able to solve all instances with 10 nodes per tree to optimality within approximately 10 seconds. No timeout occurs either in the pairwise greedy merges for these instances. As Table I shows, the greedy method performs slightly worse on these instances, yielding solutions with two additional edges on average. None of the other instances are solved to optimality by IDP; the solver either times out (2 000 seconds), or runs out of memory. Interestingly, the quality of the solutions obtained by the greedy approach vastly exceeds that of the solutions obtained by the exact solver on the largest instances in Table I. This is partly due to the large amount of memory used by the SAT solver, which keeps learning clauses as it runs; the solver eventually runs out of memory and returns the best solution found so far. Since this occurs frequently, even after running IDP for only 300 seconds, these solutions are worse than what IDP would have found in 2 000 seconds. However, this only partially explains the differences: on some instances (e.g. those with 20 trees with 5 labelled nodes), IDP does reach the 2 000 second time limit and still performs a lot worse than the greedy method. We therefore conclude that on large instances the pairwise approach is a very promising method for solving CS-PLT.

We also investigated how the loss of quality evolves with the number of trees in the input. Figure 9 compares the sizes of the solutions obtained by the exact method and the greedy methods on random instances made of $(12, 6)$-trees with no timeout. Solutions obtained by the greedy method were at most 13% larger than those obtained by the exact method.

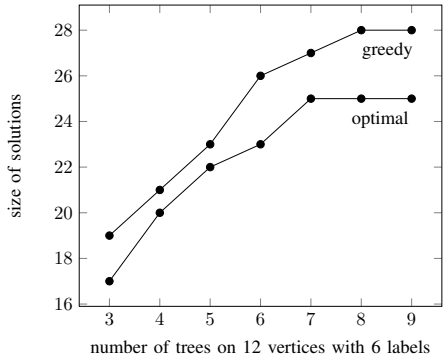

Fig. 9. Number of edges obtained by the exact and the greedy methods on random instances as the number of trees increases (no timeouts). The greedy approach produced solutions that were at most 13% larger than the optimal solution.

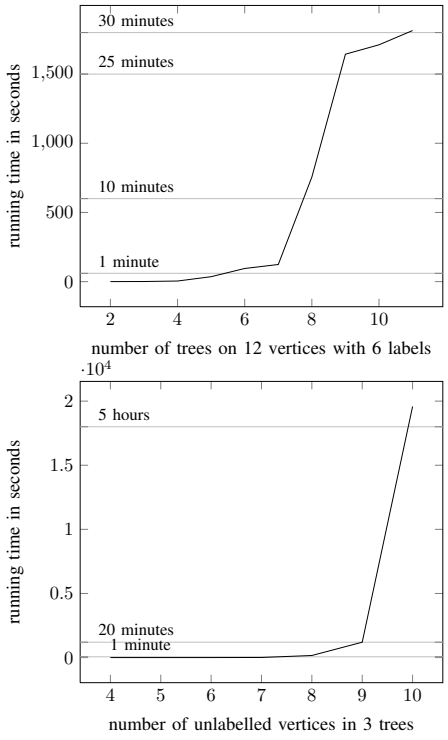

Fig. 10. Growth of the exact solver's running time with respect to the number t of trees (averages over 20 runs) or the number k of unlabelled vertices (averages over 50 runs with $n = 12$). Note that the search space has size $O((n-k)!^{t-1})$.

Figure 10 concludes our experiments and shows how the running time of the exact solver grows with respect to the instance size, measured on the one hand by the number of trees in the instance and, on the other hand, by the number of unlabelled nodes in those trees.

V. CONCLUSIONS

In this work, we have shown that the decision version of the problem of finding a minimum common supergraph of a given set of partially labelled trees is NP-complete, which justifies and magnifies the importance of good approximate solutions to the original optimisation problem, as well as fast heuristics and exact algorithms for solving it in

practice. In that regard, we have investigated how promising the popular SAT solver-based approach could be in our case; we bypassed the difficulties that arise when trying to encode instances and problem descriptions as Boolean formulas by relying on the IDP system to handle the translation to a SAT instance and then to solve instances of our problem using a SAT solver. We proposed an optimised model that allowed us to obtain both an exact solution to our problem and a greedy approach that proved very useful in practice, yielding very high quality solutions much faster than the exact approach.

Several interesting theoretical questions arise. Most notably, the complexity of CS-PLT on two partially labelled trees remains open. Moreover, the computational complexity classification of CS-PLT could perhaps be further refined: in particular, does the problem admit a c-approximation algorithm for some constant c? Are there nice parameterisations of the problem that could prove useful in practice? The excellent performance of the greedy method justifies the importance of finding efficient algorithms for the pairwise case, since merging partial solutions in a greedy fashion usually gives solutions of high quality to the general problem. In addition, it would be interesting to further investigate the case where at least one of the input graphs is a graph instead of a tree, both from a complexity point of view and from an approximation point of view.

As far as practical aspects are concerned, fast and accurate solutions for real-world instances with actual data are still needed, especially in light of the problem's complexity. Future work will in particular investigate how the SAT solver-based approach proposed in this paper applies and scales in practice.

Finally, other considerations might need to be taken into account in order to assess the relevance of the results yielded by the UMP method in practice, which will require input from biologists. Are there other parameters that should be taken into account when searching for a minimum common supergraph? Which criteria should be used to discriminate between nonisomorphic optimal solutions? We note that additional criteria could be easily incorporated directly into IDP, using the multitude of available logical operators and arithmetic operations.

Acknowledgments

We wish to thank Broes De Cat and Johan Wittocx for explanations about the IDP system and for their help in improving the model shown in Figure 5, as well as the referees for their helpful and insightful comments. The second author is supported by STW project 11763 (ITALIA) and STW project 13136 (MANTA).

References

[1] H. J. Bandelt, P. Forster, B. C. Sykes, and M. B. Richards, *Mitochondrial portraits of human populations using median networks*, Genetics, 141 (1995), pp. 743–753.

[2] A. Biere, M. Heule, H. van Maaren, and T. Walsh, *Handbook of Satisfiability*, IOS Press, 2009.

[3] I. Cassens, P. Mardulyn, and M. C. Milinkovitch, *Evaluating intraspecific "network" construction methods using simulated sequence data: Do existing algorithms outperform the global maximum parsimony approach?*, Syst. Biol., 54 (2005), pp. 363–372.

[4] S. A. Cook, *The complexity of theorem-proving procedures*, in Proc. 3rd STOC, Shaker Heights, Ohio, USA, 1971, ACM, pp. 151–158.

[5] R. Diestel, *Graph theory*, vol. 173 of Graduate Texts in Mathematics, Springer-Verlag, Berlin, third ed., 2005.

[6] J. Felsenstein, *Inferring Phylogenies*, Sinauer Associates, Sunderland, MA, 2004.

[7] P. Gambette, *Who is who in phylogenetic networks: Articles, authors and programs*. Published electronically at `http://www.atgc-montpellier.fr/phylnet`.

[8] C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman, *Handbook of Knowledge Representation*, Foundations of Artificial Intelligence, Elsevier Science, 2007, ch. Satisfiability Solvers.

[9] D. H. Huson, R. Rupp, and C. Scornavacca, *Phylogenetic Networks: Concepts, Algorithms and Applications*, Cambridge University Press, Nov. 2010.

[10] A. Labarre, *Combinatorial aspects of genome rearrangements and haplotype networks*, PhD thesis, Université Libre de Bruxelles, Brussels, Belgium, Sept. 2008.

[11] M. Mariën, J. Wittocx, M. Denecker, and M. Bruynooghe, *SAT(ID): Satisfiability of propositional logic extended with inductive definitions*, in Proc. 11th SAT, vol. 4996 of Lecture Notes in Computer Science, Guangzhou, China, May 2008, Springer, pp. 211–224.

[12] T. J. Schaefer, *The complexity of satisfiability problems*, in Proc. 10th STOC, San Diego, California, USA, May 1978, ACM, pp. 216–226.

[13] J. Wittocx, M. Mariën, and M. Denecker, *The IDP system: a model expansion system for an extension of classical logic*, in Proc. 2nd LaSh, Leuven, Belgium, Nov. 2008, pp. 153–165.

[14] J. Wittocx, M. Mariën, and M. Denecker, *Grounding FO and FO(ID) with bounds*, J. Artificial Intelligence Res., 38 (2010), pp. 223–269.

Anthony Labarre obtained his PhD degree in Computer Science in 2008 from the Université Libre de Bruxelles, Brussels, Belgium. He is currently an Associate Professor at the Université Paris-Est Marne-la-Vallée (France). His research interests include genome rearrangements, phylogenetic networks, algorithms and enumerative combinatorics.

Dr. ir. Sicco Verwer is assistant professor computer science at Delft University of Technology. After receiving his PhD. degree from the same university, he spent time as a postdoctoral researcher at Eindhoven University of Technology, Catholic University Leuven, and Radboud University Nijmegen, and as a researcher at the ministry of Security and Justice in the Netherlands. His research focuses on the theory and practice of machine learning, and state machine learning in particular. Dr. Verwers interests within this focus area are diverse. He has published papers on learning state machines, discrimination-aware data mining, software testing, fraud detection, mechanism design, and combinatorial solvers in machine learning. In particular, dr. Verwer is interested in machine learning algorithms resulting in models that are useful for subsequent tasks such as visualization, testing, verification, data integration, control, and optimization. The focus of his work is in the cyber-security domain where he aims to learn models of communication protocols from streaming network traffic.