Algorithmic investigations of the dynamics of species interactions

Yishu Wang

N° d'ordre NNT : 2021LYSE1196

THÈSE de DOCTORAT DE L'UNIVERSITÉ DE LYON
opérée au sein de
l'Université Claude Bernard Lyon 1

École Doctorale N° 341
Évolution, Écosystème, Microbiologie, Modélisation

Spécialité de doctorat : Bioinformatique

Soutenue publiquement le 05/10/2021, par :

Yishu Wang

Algorithmic investigations of the dynamics of species interactions

---

Devant le jury composé de :

Lobry, Jean	PR, Université Claude Bernard Lyon 1	Président
Middendorf, Martin	PR, Leipzig University	Rapporteur
Semple, Charles	PR, University of Canterbury	Rapporteur
Finocchi, Irene	PR, LUISS University	Examinatrice
Strozecki, Yann	MCF, Université UVSQ	Examinateur
Sagot, Marie-France	DR, INRIA	Directrice de thèse
Figueiredo, Mário	PR, Instituto Superior Técnico	Co-directeur de thèse
Sinaimeri, Blerina	CR, INRIA	Co-encadrante de thèse
Mary, Arnaud	MCF, Université Claude Bernard Lyon 1	Invité

Université Claude Bernard – LYON 1

Président de l'Université	M. Frédéric FLEURY
Président du Conseil Académique	M. Hamda BEN HADID
Vice-Président du Conseil d'Administration	M. Didier REVEL
Vice-Président du Conseil des Études et de la Vie Universitaire	M. Philippe CHEVALLIER
Vice-Président de la Commission de Recherche	M. Jean-François MORNEX
Directeur Général des Services	M. Pierre ROLLAND

COMPOSANTES SANTÉ

Département de Formation et Centre de Recherche en Biologie Humaine	Directrice : Mme Anne-Marie SCHOTT
Faculté d'Odontologie	Doyenne : Mme Dominique SEUX
Faculté de Médecine et Maïeutique Lyon Sud - Charles Mérieux	Doyenne : Mme Carole BURILLON
Faculté de Médecine Lyon-Est	Doyen : M. Gilles RODE
Institut des Sciences et Techniques de la Réadaptation (ISTR)	Directeur : M. Xavier PERROT
Institut des Sciences Pharmaceutiques et Biologiques (ISBP)	Directrice : Mme Christine VINCIGUERRA

COMPOSANTES & DÉPARTEMENTS DE SCIENCES & TECHNOLOGIE

Département Génie Électrique et des Procédés (GEP)	Directrice : Mme Rosaria FERRIGNO
Département Informatique	Directeur : M. Behzad SHARIAT
Département Mécanique	Directeur M. Marc BUFFAT
École Supérieure de Chimie, Physique, Électronique (CPE Lyon)	Directeur : M. Gérard PIGNAULT
Institut de Science Financière et d'Assurances (ISFA)	Directeur : M. Nicolas LEBOISNE
Institut National du Professorat et de l'Éducation	Administrateur Provisoire : M. Pierre CHAREYRON
Institut Universitaire de Technologie de Lyon 1	Directeur : M. Christophe VITON
Observatoire de Lyon	Directrice : Mme Isabelle DANIEL
Polytechnique Lyon	Directeur : M. Emmanuel PERRIN
UFR Biosciences	Administratrice provisoire : Mme Kathrin GIESELER
UFR des Sciences et Techniques des Activités Physiques et Sportives (STAPS)	Directeur : M. Yannick VANPOULLE
UFR Faculté des Sciences	Directeur : M. Bruno ANDRIOLETTI

Abstract

To understand the dynamics of interaction between groups of species, for example, in a hosts/parasites (or hosts/symbionts) system, it is essential to consider the process of coevolution. The cophylogeny reconciliation model formulates the coevolution question as an optimization problem. The set of optimal solutions represents different scenarios of coevolution which need to be analyzed separately. The main result of this thesis is a novel approach that addresses the issue of the often huge number of optimal solutions which makes it difficult to analyze the results. We introduce several biologically motivated equivalence relations, each one splitting the solution space into equivalence classes. We propose polynomial-delay algorithms that enumerate the equivalence classes, and the representative solutions, i.e., the solutions taken from each equivalence class. Experimental results are then presented to illustrate the practical benefits of considering equivalence classes as a means of efficiently exploring the solution space of cophylogeny reconciliation. Based on the algorithmic results, a software called Capybara has been developed as a practical tool for analyzing cophylogeny datasets. More generally, in the area of enumeration algorithms, the problem of efficiently enumerating representative solutions is of theoretical importance. In the second part of this thesis, we provide a general framework for the enumeration of the equivalence classes of solutions for a family of optimization problems. We show that this framework can be applied to various dynamic programming problems, of which the cophylogeny problem is a special case.

Keywords.
Cophylogeny, Enumeration algorithms, Equivalence relation, Dynamic programming.

Résumé

Pour comprendre la dynamique d'interactions entre des groupes d'espèces, par exemple, dans un contexte hôtes-parasites, il est primordial d'étudier le processus de coévolution. La réconciliation cophylogénétique est un modèle qui traduit la reconstruction de l'histoire coévolutive en un problème d'optimisation combinatoire. L'ensemble des solutions optimales, qui représentent des scénarios coévolutifs différents, peut avoir une taille considérable, rendant l'analyse des résultats difficile. Le résultat principal de cette thèse porte sur une nouvelle méthode qui permet d'explorer l'espace de solutions de manière efficace. D'abord, je définis des relations d'équivalence et propose des algorithmes d'énumération qui produisent la liste des solutions représentatives, c'est-à-dire des solutions appartenant à chacune des classes d'équivalence. Ensuite, je présente des résultats expérimentaux et montre que l'analyse des classes d'équivalence aide à mieux étudier des jeux de données biologiques. Basé sur nos résultats algorithmiques, un logiciel appelé Capybara a été développé comme un nouvel outil pratique pour l'analyse cophylogénétique. Plus généralement, l'énumération des classes d'équivalence des solutions est un problème théorique important. Dans la seconde partie de la thèse, je présente un cadre général au sein duquel l'énumération efficace des classes d'équivalence est possible. Je propose un algorithme qui énumère les classes d'équivalence des solutions pour une famille de problèmes, en particulier, des problèmes où s'applique la technique de programmation dynamique.

Mots-clés.
Cophylogénie, Algorithmes d'énumération, Relation d'équivalence, Programmation dynamique.

Résumé en français

Le lecteur avisé ne devrait pas constater avec surprise que, contrairement à ce qu'un titre de « dynamiques des interactions inter-espèces » aurait suggéré, le sujet principal de cette thèse concerne non pas l'écologie mais l'évolution. En effet, des approches mêlant écologie et évolution ont été développées notamment par des écologues qui s'intéressent à l'impact des mécanismes coévolutifs sur des communautés biologiques en étroite interaction.

Pour préserver la biodiversité ou pour maîtriser l'émergence de maladies infectieuses, les sciences de l'écologie doivent répondre au défi de la prédiction. Puisqu'il est impossible de concevoir un avenir sans une compréhension approfondie du passé, une science prédictive, quelle que soit l'échelle temporelle visée, s'appuie nécessairement sur les sciences du passé : géologie, paléontologie, archéologie, histoire, ou encore, la biologie évolutive. La cophylogénie, la juxtaposition des deux arbres phylogénétiques, consiste à reconstruire l'histoire évolutive conjointe de deux groupes d'organismes ayant à l'heure actuelle une certaine relation écologique entre eux, par exemple un groupe de mammifères et un groupe de leurs parasites. Il s'agit alors d'établir deux types de liens de façon simultanée : des liens écologiques entre ces deux groupes d'espèces, ce qui délimite le périmètre de l'étude, et des liens évolutifs entre le présent et le passé, ce qui accorde aux résultats un pouvoir informatif et prédictif.

Il est facile de représenter des liens entre des organismes vivants : connaissant quels parasites vivent chez quels hôtes, nous pouvons simplement écrire leurs noms sur un papier et les relier par des traits de crayon. Sur l'axe temporel, comment représenter des liens évolutifs entre le présent et le passé ?

La philosophie de la Grèce antique s'offre deux temporalités différentes : Chronos et

Kairos. Chronos fait référence au temps qui défile de manière linéaire et séquentielle, tandis que Kairos symbolise le temps du moment opportun pour un geste décisif. Un arbre phylogénétique, avec ses nœuds à chaque ramification, incarne Kairos, les points de basculement où de nouvelles espèces sont nées. De même, la cophylogénie cherche à établir des liens évolutifs qui ne s'inscrivent pas dans une temporalité chronologique, mais qui peuvent être identifiés par des *événements coévolutifs*, une sorte de transition critique durant laquelle se forment de nouvelles associations écologiques.

Le problème cophylogénétique est le suivant. Nous disposons de deux arbres phylogénétiques (qui représentent l'histoire évolutive de deux groupes d'espèces, comme celui d'hôtes et celui de parasites), et d'un ensemble d'associations qui indiquent quel organisme d'un arbre interagit au temps présent avec quel organisme de l'autre arbre. L'objectif est d'associer les espèces ancestrales entre les deux arbres, d'une façon qui soit la plus judicieuse possible, et ce, en termes d'événements coévolutifs représentés par ces associations.

Le modèle de *réconciliation d'arbres*, proposé par des bioinformaticiens, permet de transformer le problème cophylogénétique en un problème d'optimisation combinatoire. Pour trouver une solution optimale d'un problème d'optimisation, le développement d'algorithmes est un outil essentiel. S'attaquant à ce problème informatique, notre investigation est donc de nature algorithmique.

Un algorithme, une suite finie et non ambiguë d'instructions qui peuvent s'exécuter sur un ordinateur, apporte une réponse à la question cophylogénétique au moyen d'un autre type de lien plus abstrait. Entre les liens écologiques inter-espèces et les liens évolutifs composés d'événements coévolutifs, un algorithme forme le maillon central qui relie deux *espaces* : l'un est l'espace d'hypothèses biologiquement plausibles de l'histoire coévolutive, l'autre est l'espace de solutions optimales du problème informatique.

La vraisemblance d'une hypothèse coévolutive se rapporte à une pluralité de réflexions. Or, un modèle mathématique ne peut prendre en compte qu'un petit nombre d'entre elles. En outre, l'inclusion de certaines contraintes biologiques pourrait rendre le problème informatique difficile à résoudre, ou, comme les informaticiens l'appellent,

NP-difficile. La réponse apportée par un algorithme ne peut donc pas être une réponse directe à la question biologique. Si l'espace des hypothèses biologiques n'est jamais identique à celui des solutions du problème informatique, le maillon algorithmique entre ces deux espaces repose sur une intime conviction des bioinformaticiens : avec un modèle soigneusement élaboré, la vérité biologique est intégralement saisie, et il ne reste plus qu'à la localiser au sein de l'espace de solutions, c'est-à-dire, dans la sortie de l'algorithme. En d'autres termes, si un algorithme pouvait fournir une représentation exhaustive de l'espace de solutions, une analyse appropriée de sa sortie permettrait de dévoiler l'ensemble des scénarios coévolutifs plausibles.

Ainsi, des algorithmes *d'énumération* ont été développés pour le problème de cophylogénie. Un tel algorithme produit une liste contenant différentes manières d'associer les deux arbres phylogénétiques, toutes optimales par rapport aux contraintes mathématiques basées sur les événements coévolutifs. Un biologiste peut alors analyser chacune de ces solutions, appliquer des critères de sélection supplémentaires, et arriver à un espace d'hypothèses coévolutives appuyé par son expertise.

La notion d'espace de solutions joue un rôle clé dans notre investigation algorithmique de la cophylogénie et marque le point d'entrée de cette thèse. Bien qu'un algorithme d'énumération efficace semble donner une réponse satisfaisante à la question biologique, il y a toutefois un obstacle de taille : il s'agit de la *taille* de l'espace de solutions, c'est-à-dire du nombre de solutions optimales qui le composent. Afin de soumettre la liste de solutions à un biologiste qui procédera ensuite à l'analyse manuelle des résultats, ce nombre-là doit rester raisonnable. Au cas où il y aurait plusieurs milliers de solutions, la tâche d'analyse de résultats pourrait être confiée à un programme informatique. Cependant, aucun programme informatique ne pourra jamais traiter une liste contenant un si grand nombre d'objets qu'il dépasse le nombre d'atomes dans l'univers ![1]

L'espace de solutions pour le problème de cophylogénie peut, malheureusement,

[1] Dans le Chapitre 3, le plus grand espace de solutions que nous allons rencontrer est de l'ordre de 10^{136}. Le nombre d'atomes dans l'univers a un ordre de grandeur estimé entre 10^{78} et 10^{82} (une brève explication de cette estimation se trouve au lien suivant : https://physics.stackexchange.com/questions/47941/dumbed-down-explanation-how-scientists-know-the-number-of-atoms-in-the-universe).

devenir incroyablement vaste, et surtout si on s'intéresse à un système d'interactions hôtes-parasites impliquant des milliers d'organismes. Que faire dans ce cas-là ?

Une stratégie possible consiste à affiner le modèle en y ajoutant des contraintes supplémentaires. Davantage de contraintes pourront effectivement réduire la taille de l'espace de solutions. Néanmoins, comme nous l'avons évoqué plus tôt, il se peut que ces contraintes rendent le problème bien plus difficile à résoudre, jusqu'au point où l'on se retrouve dans l'incapacité de concevoir un algorithme efficace. De plus, de nouvelles contraintes exigent de nouvelles informations biologiques (e.g., des connaissances biogéographiques qui excluent l'association entre des espèces spatialement séparées), qui peuvent s'avérer difficiles à obtenir ou insuffisamment fiables.

Le travail de cette thèse relève d'une autre stratégie : naviguer à travers un vaste espace de solutions de manière efficace. Lorsqu'il est irréaliste de faire un parcours exhaustif de la longue liste de solutions, un « parcours guidé » de quelques solutions représentatives laisserait peut-être déjà se dessiner le contour des scénarios coévolutifs les plus vraisemblables.

Dans le Chapitre 1, je présente brièvement quelques concepts biologiques et mathématiques et introduis le défi de trouver des solutions représentatives du problème de cophylogénie. Le Chapitre 2 est consacré aux principaux résultats algorithmiques. D'abord, je définis plusieurs relations d'équivalence sur l'ensemble de solutions optimales et explique la motivation biologique de ces définitions. Mathématiquement, une relation d'équivalence partitionne l'espace de solutions en des sous-parties appelées « classes d'équivalence ». Dans ce contexte, on considère comme étant représentatives les solutions, non équivalentes entre elles, qui appartiennent à chacune des classes d'équivalence. Je présente ensuite des algorithmes permettant d'énumérer les classes d'équivalence pour chaque relation d'équivalence. Afin d'illustrer les avantages de considérer les classes d'équivalence en pratique, des résultats expérimentaux sur des jeux de données biologiques sont donnés dans le Chapitre 3. Ce chapitre présente aussi le logiciel Capybara, un outil d'aide à l'analyse des données de cophylogénie qui dérive de notre méthode d'exploration de l'espace de solutions par le biais des classes d'équivalence.

Notre investigation algorithmique du problème cophylogénétique nous mène à réaliser deux autres investigations liées à la première. Dans le Chapitre 4, on se propose de s'attaquer à la question d'énumérer les classes d'équivalence de solutions dans un contexte plus général, où une solution est identifiée par un sous-arbre dans un graphe, et la relation d'équivalence est donnée par un coloriage de ce graphe. Je présente un algorithme d'énumération de classes d'équivalence et montre que celui-ci pourrait s'appliquer à une famille de problèmes différents, dont fait partie le problème de cophylogénie. Le Chapitre 5 aborde la question de la quantification des différences entre des arbres phylogénétiques. De nombreuses applications nécessitent un calcul rapide et précis d'une distance entre des arbres. Par exemple, dans un système d'interactions hôtes-parasites soumis à une forte pression favorisant la coévolution, on s'attend à constater une faible dissimilarité entre les deux arbres phylogénétiques. Je propose un nouvel algorithme pour approximer la distance dite *Subtree prune and regraft* entre deux arbres, et je réalise une série d'expériences visant à illustrer ses avantages pratiques par rapport aux algorithmes existants.

Les principaux résultats algorithmiques et une partie des résultats expérimentaux des Chapitres 2 et 3 se trouvent dans l'article « Making sense of a cophylogeny output : Efficient listing of representative reconciliations », publié dans le *Workshop on Algorithms in Bioinformatics* (WABI) 2021 [Wan+21b].

Le logiciel Capybara a été présenté dans l'article « Capybara : equivalence ClAss enumeration of coPhylogenY event-BAsed ReconciliAtions », publié sous forme d'une Application Note dans la revue *Bioinformatics* [Wan+20].

Le contenu du Chapitre 4 constitue l'article « A general framework for enumerating equivalence classes of solutions », publié dans l'European Symposium on Algorithms (ESA) 2021 [Wan+21a].

Acknowledgements

Throughout the preparation and the writing of this thesis, I have received a great deal of support and assistance from many people. As someone who is quite uncomfortable with names (even with my own!), I would avoid using names as much as possible (the same applies to the rest of the manuscript).

I would like to express my deep gratitude to my research supervisors, for their patient guidance, enthusiastic encouragement and invaluable critiques of this research work. I would also like to thank the members of the defense jury, the thesis committee, and the administrative units from the lab and the university.

I would like to acknowledge all the colleagues with whom I worked in the same office, the same team, the same lab, the same building, the same campus. I should also appreciate my confrères outside of academia, collaborators, family members, partners, companions, and those whom I met in one of the many serendipitous encounters. This thesis would not have been done in the way that it has been done without the wonderful human environment in which I continue to evolve.

Table of contents

List of Figures

List of Tables

List of symbols and abbreviations

ch	The set or tuple of children of a node.
DAG	Directed acyclic graph.
$v \nsim w$	Two nodes are incomparable.
$d_T(v, w)$	Distance between two nodes in a tree.
$LCA(v, w)$	Lowest common ancestor of two nodes in a tree.
$T\|_v$	Subtree rooted at a node.
$F_1 \cong F_2$	Isomorphic phylogenetic forests.
DP	Dynamic programming.
$\vec{e}(\phi)$	The event vector of a reconciliation.
$\mathcal{T}(G)$	The set of solution subtrees of an ad-AND/OR graph.
gch	The tuple of grandchildren of a node.
$\pi(\mathcal{T}(G))$	The set of equivalence classes of a reconciliation graph.
$\mathcal{C}(G)$	The set of color classes of an e-colored ad-AND/OR graph.
AF	Agreement forest.
MAF	Maximum agreement forest.
CS	Continuous subtree.
CCS	Common continuous subtree.

Introduction

All my apologies to the reader who, intrigued by the title of *dynamics of species interactions* and eager to catch a glimpse of the complex mechanisms within ecological communities, will discover with surprise that most of this thesis is concerned with evolution rather than ecology. Yet, it is a hallmark of the modern-day ecology to consider the process of *coevolution* for the purpose of investigating biodiversity and certain aspects of disease ecology.

To understand the present and to predict the future, humankind possesses an essential source of knowledge: the past. Just as geology, paleontology, archaeology or history, evolutionary biology makes it possible to establish a link between the present time and a bygone time, either distant or nearby. In cophylogeny, evolution is studied with regard to two taxonomic groups assumed to have some kind of current relationship with each other, such as where one is a group of mammal species and the other is a group of parasites of those mammals. Cophylogeny thus tries to elucidate two links simultaneously: the ecological link between the two groups of species, which sets the perimeter of the investigation, and the evolutionary link between the present and the past, which endows the results with informative and predictive power.

To make the link between two groups of species is easy: after observing which parasites live in which mammals, we can simply write down their names on a paper and draw lines between them with a pencil. But how do we connect the present and the past?

The ancient Greeks had two words for time: chronos and kairos. Chronos refers to sequential time, while kairos signifies a proper time for a decisive act. A phylogenetic tree, with its nodes connecting separate branches, embodies kairos, the tipping points

where new species are born. Likewise, the evolutionary link made by cophylogeny is not chronological in nature, but is identified by *coevolutionary events*, the critical transitions during which new ecological connections are formed.

The cophylogeny problem is the following. Our information thus far has three components: the two phylogenetic trees (representing respectively the evolutionary history of a group of host species and of a group of parasite species), and a set of associations that state which taxon in one tree is associated with which taxon in the other tree. Our objective is to find a way of associating the ancestral species in the parasite evolutionary tree with places in the host evolutionary tree, in a manner that makes the most sense, and we judge what makes sense in terms of the coevolutionary events that would account for those past associations.

Formulated as a computational problem, the phylogenetic tree reconciliation model translates the cophylogeny problem into a hunt for efficient algorithmic tools. Our investigation is therefore an algorithmic one.

Algorithms, finite sequences of well-defined and computer-implementable instructions, help answering the cophylogeny question by providing another, more abstract, kind of link. Built on top of the ecological link between species and the evolutionary link composed of coevolutionary events, the algorithmic link connects *spaces*: one is the space of biologically plausible hypotheses on associations and events, which forms the answer of the biological question, the other one is the space of optimal solutions of the computational problem, which is a mathematically well-defined set, and can constitute the output of a computer program.

For a hypothesis about the coevolutionary history to be biologically plausible, a broad range of factors should be taken into account, although only a select few can be encoded as mathematical constraints into the computational problem. Besides, there are biological constraints that, once added to the computational problem, render it intractable, or, as the computer scientists call it, NP-hard. Needless to say, an algorithm does not directly answer the biological question: even with the most accurate mathematical model, the space of biological hypotheses is never equivalent to the space of solutions of the computational problem. By saying that there is an algorithmic link be-

tween the two spaces, we express our belief that, given a sufficiently accurate model, biological insights should be hidden somewhere in the *solution space*, and hence could be found in the output of the algorithm. In other words, an algorithm that supplies the whole solution space should open the door to understanding different possibilities of coevolutionary scenarios.

This is the case for *enumeration algorithms* that have been developed for the cophylogeny problem. Such an algorithm enumerates, that is, provides a list of different ways of associating the two phylogenetic trees, all of which are optimal with respect to mathematical constraints based on the occurrences of coevolutionary events. Then, a biologist can analyze each one of these solutions, apply additional selection criteria, and come up with her own space of biologically plausible coevolutionary hypotheses.

Indeed, the solution space plays a pivotal part in our algorithmic investigation of the cophylogeny problem, and is the starting point of this thesis. It may seem that using an efficient enumeration algorithm we can already give a satisfactory answer to the biological question. There is nonetheless a stumbling block: the *size* of the solution space. For a human expert to be able to analyze the solutions one by one, the number of solutions cannot be too large. When there are thousands of solutions, we can resort to computer programs that automatically perform the analysis. However, no computer program can ever process a list containing such a huge number of items that it exceeds the number of atoms in the observable universe! [2]

For the cophylogeny problem, the size of the solution space does unfortunately get incredibly large, especially if we want to study a hosts/parasites system composed of thousands of organisms. What can we do in such cases?

One possible strategy is to refine the model by including more constraints. This new version of the computational problem will necessarily have a smaller solution space. We already mentioned one major drawback of this approach: adding more constraints can make the problem harder to solve, and we may be unable to find efficient algo-

[2] In Chapter 3, the largest solution space that we will encounter has a size in the order of 10^{136}. It is estimated that the there are between 10^{78} to 10^{82} atoms in the observable universe (a short explanation of this estimation can be found in https://physics.stackexchange.com/questions/47941/dumbed-down-explanation-how-scientists-know-the-number-of-atoms-in-the-universe).

rithms. Those constraints also require further biological information (such as biogeographical information that makes associations between spatially separated species impossible), which can be difficult to obtain or insufficiently reliable.

Another strategy consists in exploring the solution space efficiently despite its large size. Many efforts have been undertaken in this direction. The work of the current thesis also lies within the scope of this approach.

In Chapter 1, along with a succinct presentation of the mathematical and biological background, we introduce our main algorithmic challenge: finding representative solutions of the cophylogeny problem. Chapter 2 sets out the formal definition of the model and our theoretical results. First, we define equivalence relations on the set of solutions and explain the biological motivation of the definitions. Mathematically, an equivalence relation splits the solution space into parts of mutually equivalent subsets, called *equivalence classes*. In this setting, we find representative solutions through the search of solutions that are not equivalent to each other, i.e., solutions in different equivalence classes. We propose algorithms for enumerating the equivalence classes, for each notion of equivalence relation. In Chapter 3, we show the experimental results that illustrate the practical benefits of using the equivalence classes as a means of exploring the solution space. A software called Capybara is also presented in Chapter 3 as a practical tool for analyzing cophylogeny datasets. Then, we conclude our work on the cophylogeny problem and present some perspectives.

Our algorithmic investigation of the cophylogeny problem has fostered two related investigations. In Chapter 4, we venture into the territory of the abstract: in a general framework where the solution space of some problem needs to explored, we propose an algorithm for enumerating the equivalence classes of solutions, with respect to an equivalence relation represented as a graph coloring. We show that the algorithm can be applied to various computational problems, of which the cophylogeny problem is a special case. In Chapter 5, we turn our attention to another problem in phylogenetic studies that has to do with the quantification of dissimilarity between phylogenetic trees. A fast and accurate computation of a distance measure between trees has many applications. For instance, in a hosts/parasites system where species have co-

evolved together, the dissimilarity between the phylogenetic trees of the two groups of species should be low. We propose a new algorithm for approximating the subtree prune-and-regraft distance between two phylogenetic trees, and we perform a series of experiments to demonstrate its practical advantages in comparison with existing algorithms.

The main algorithmic results and some of the experimental results from Chapters 2 and 3 are included in the paper "Making sense of a cophylogeny output: Efficient listing of representative reconciliations", published in the *Workshop on Algorithms in Bioinformatics* (WABI) 2021 [Wan+21b].

The cophylogeny software Capybara has been presented in "Capybara: equivalence ClAss enumeration of coPhylogenY event-BAsed ReconciliAtions", published as an application note in the journal *Bioinformatics* [Wan+20].

The material in Chapter 4 forms the paper "A general framework for enumerating equivalence classes of solutions", published in *European Symposium on Algorithms* (ESA) 2021 [Wan+21a].

Chapter 1

Preliminaries

1.1 Biological background

1.1.1 Phylogenetics

A central theme in the study of evolution is the understanding of evolutionary relationships between organisms, both living and extinct. To describe such relationships, a conceptual tool, dating back to the mid-nineteenth century, is the so-called *Tree of Life*. Charles Darwin in his famous book *The Origin of Species* presented a first graphical representation of evolutionary relationship among species, in the form of a tree [Dar88]. In evolutionary biology, phylogenies, or phylogenetic trees, have been a model of choice, on the tenet that contemporary species all share a common history through their ancestry [DBP05].

As little is known about the relationships between extinct species, phylogenetic trees are *reconstructed* by using phylogenetic *inference methods*, where the past relationships are inferred through mathematical models from features of contemporary species. Early methods for reconstructing phylogenetic trees relied on morphological or structural characteristics. Today, advances in molecular biology, especially in sequencing technologies, have allowed the extensive use of DNA or protein sequence data to build phylogenetic trees.

Phylogenetic inference methods that use molecular data are based on the identification of homologous genes, that is, genes from different organisms that have a shared ancestry [Fit70]. The development of new sequencing technologies has given rise to massive datasets of ever increasing size, both in the number of genes and in the number of taxa (i.e., species) that can be considered [KYT20]. Consequently, more sophisticated methods have been developed to extract the most information out of those datasets, resulting in more accurate trees on larger and larger scales [Hug+16; Bur+20; CK20]. Figure 1.1 shows, for example, the tree of life constructed by Hug et al. [Hug+16] using genomic data from 3083 organisms.

As the volume of the data is huge and complex methods are required to deal with the uncertainties of the data and systematic errors (these errors are generally due to heterogeneous rates of evolution across taxa or time, which violate the assumptions of the models), modern phylogenetic inference methods pose a heavy computational burden. The inference of the tree in Figure 1.1 required 3840 computational hours on a supercomputer.

Algorithmic efficiency is thus crucial. In Chapter 5, from an algorithmic standpoint, we will look at one problem that naturally arises in phylogenetics analysis, that is, the quantification of the difference between phylogenetic trees.

1.1.2 Cophylogeny

In cophylogeny, one is interested in reconstructing the *coevolutionary history* of groups of species that interact over a long period of time and in ecologically close environments based on their phylogenetic information. This prolonged and intimate interaction between different species is known as *symbiosis*. Symbioses are a common feature of life and can be found across all major phylogenetic lineages [Pou11]. Typically defined between a larger host organism and a smaller symbiont, symbiotic relationships may be *mutualistic*, where both partners benefit from the interaction, *parasitic*, where one partner benefits and the other suffers a cost, or *commensalistic*, where one partner benefits while the other receives neither benefit nor harm [LP08; EKM16].

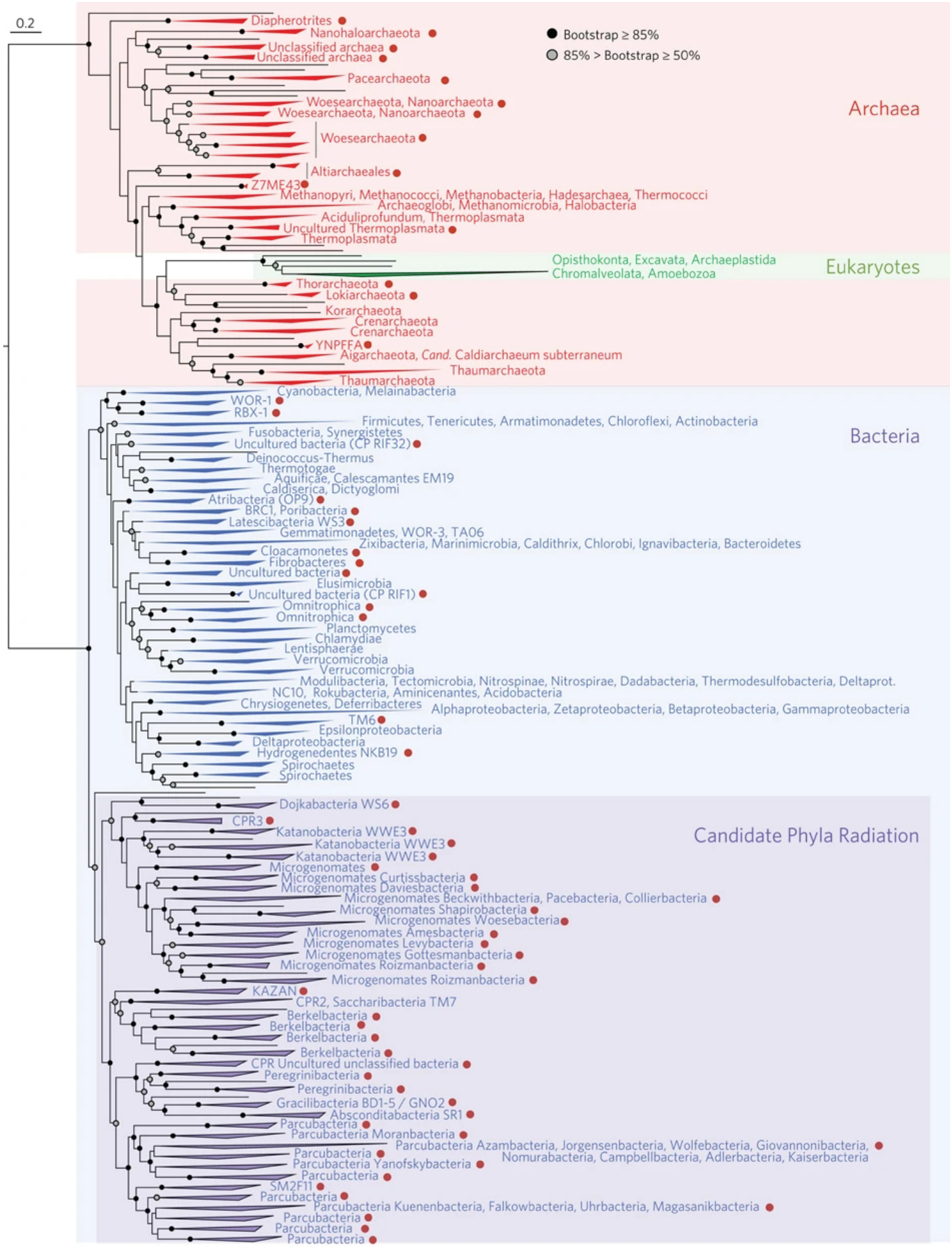


Figure 1.1: The tree of life constructed from genomic data by Hug et al. [Hug+16].

Understanding the coevolutioary history of groups of organisms is important for many reasons, including the identification and tracing of the origins of emerging infectious diseases [Eth+06; LO14; Pen+15], and the conservation and management of animal or microbial diversity [Ley+08; Bah+16].

Chapters 2 and 3 are dedicated to *phylogenetic tree reconciliation*, a widely used method in cophylogeny studies. Following a long list of papers that considered this method [Pag94; Cha98; MM05; Con+10; Don+15], we will use the terminology in the context of hosts/parasites (hosts/symbionts) systems, even though the method can also be used in the closely related context of genes/species coevolution [THL11; BAK12; Doy+10; Sto+12]. In a nutshell, doing a phylogenetic tree reconciliation means to *map* the parasite tree to the host tree. Once an appropriate mapping between the two trees is chosen, one should be able to identify *coevolutionary events*. In general, four main evolutionary events are considered: cospeciation (when both parasite and host speciate), duplication (when the parasite speciates but not the host), loss (when the host speciates but not the parasite), and host-switch (when the parasite speciates and one of the new species goes to infect another unrelated host). A schematic representation of each of the four types of events is given in Figure 1.2.

Formulated as a computational optimization problem, the phylogenetic tree reconciliation model includes a cost assigned to each type of events. The goal is then to find a most parsimonious reconciliation, i.e., a mapping between the two trees that minimizes the total cost of the events.

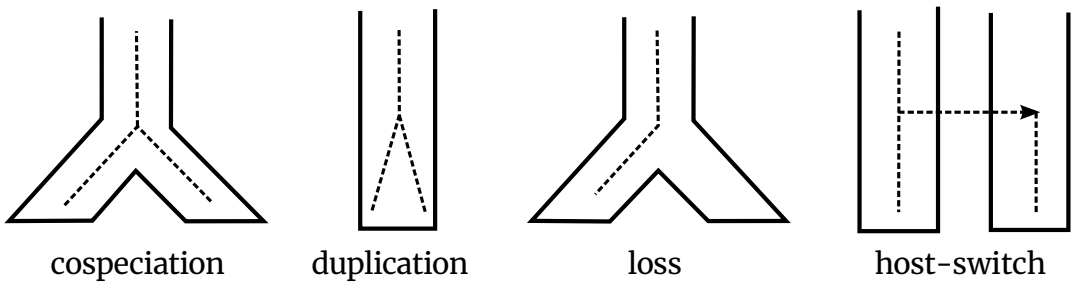


Figure 1.2: Illustration of the four types of coevolutionary events in a host/parasite reconciliation. The tube represents the host tree; the dotted line represents the parasite tree.

If timing information (i.e., the order in which the speciation events occurred in the host phylogeny) is not available (or sufficiently reliable to be used with enough confidence), as is usually the case, finding a most parsimonious reconciliation is a NP-hard problem [Ova+11; THL11]. To deal with this, one can either rely on heuristics [Con+10], or accept solutions that may be biologically unfeasible, that is, to accept reconciliations in which some of the host-switches induce a contradictory time ordering on the host species [BAK12; Don+15]. Following this second path, efficient algorithms have been proposed to *enumerate* optimal reconciliations. The output of those algorithm consists of not one but a list of reconciliations that all achieve the minimal cost within the model.

One issue that arises then is the size of the output. As the output of the enumeration algorithm can be composed of billions of solutions, there is no hope of analyzing each one of them separately. For instance, for the dataset of *Wolbachia* and their arthropod hosts, collected by the Biometry and Evolutionary Biology Laboratory [Sim+11; Sim12], the number of optimal reconciliations is unrealistically large (e.g., in the order of 10^{42}, see Chapter 3).

The good news is that, even when the number of solutions is huge, the set of solutions can be compactly represented in a graph structure. By carefully examining this graph, we can gain insight into the *solution space* and obtain information about the coevolutionary history of the species. Various approaches have been developed in the literature that aim at efficiently exploring the solution space or computing some characteristics on the set of solutions, including data analysis methods (e.g., sampling solutions uniformly at random [BAK13; Don+15], clustering [Ozd+17; ML19; San+20]) and exact methods (e.g., computing the diameter of the solution space [Haa+19], computing the distribution of the pairwise distances [SML19], computing a median reconciliation [Ngu+13]).

In Chapter 2, we propose a new exact method that allows to efficiently enumerate *representative solutions* based on the biologically motivated notion of *equivalence*: the output of the algorithm is then a list of reconciliations that are biologically non-equivalent to each other. In Chapter 3, we will see that in practice, by only considering

the representative solutions, we effectively reduce the number of solutions that need to be analyzed while still maintaining important biological information about the solution space (for example, *Is there a high incidence of cospeciations? In which branches of the parasite tree have there probably been host-switches?*).

1.2 Mathematical background

1.2.1 Graphs, trees, forests

Graphs

Throughout the manuscript, we will use several basic notions in graph theory such as cyclicity and connectivity. The definitions can be found in any classical textbook such as [Die05]. For a directed graph G, we denote by $V(G)$ and $A(G)$ respectively the set of nodes and the set of arcs of G. An arc is denoted by an ordered pair of nodes (v, w). The set of out-neighbors of a node v is called its set of *children* and is denoted $\boldsymbol{ch}(v)$. If $w \in \boldsymbol{ch}(v)$, the node w is a *child* of v. A *directed acyclic graph* (DAG) is a directed graph without any directed cycle. A directed tree is a DAG whose underlying undirected graph is acyclic and connected.

Trees

In this manuscript, a *tree* will refer to a rooted directed tree in which arcs are directed away from the root. Any such tree T has the following property: (1) exactly one node, called the root of T and denoted by $\boldsymbol{r}(T)$, does not have any in-neighbor; (2) any other node v has exactly one in-neighbor in T; this unique in-neighbor is called the *parent* of v and is denoted by $\boldsymbol{p}(v)$. A node without any child is called a *leaf*. We denote by $L(T)$ the set of leaf nodes of T. The non-leaf nodes, i.e., nodes which are not leaf nodes, are called the *internal nodes* of T.

In a tree T, if there exists a directed path from a node v to a node w, the node w is called a *descendant* of v, and v is called an *ancestor* of w; if moreover $v \neq w$, we say that w is a *proper descendant* of v, and that v is a *proper ancestor* of w. If neither w is an ancestor

of v nor w is an ancestor of v, we say that the two nodes are *incomparable*, and denote it $v \nsim w$. When v and w are comparable (i.e., they are not incomparable), the number of arcs on the directed path between v and w is called the *distance* between v and w, and is denoted by $d_T(v, w)$ (it is a non-negative integer). The largest distance between a node v and a leaf of T is called the *height of the node* v. The leaf nodes have height zero. We denote by $LCA(v, w)$ the *lowest common ancestor* of two nodes v and w, that is, the node having the smallest height among all nodes which are ancestors of both v and w. The *height of a tree* T is defined to be the height of its root node. A tree having one single node (its root is also a leaf) has height zero.

When T is a tree and v is a node of T, the *subtree of* T *rooted at* v is the subtree of T containing all descendants of v, and is denoted by $T|_v$. Clearly, $T|_v$ is a rooted tree, and its root is $r(T|_v) = v$. This way of taking subtrees does not change the height of the nodes: for any node w of $T|_v$, its height in $T|_v$ is the same as its height in the original tree T.

Forests

In this manuscript, a *forest* will always refer to a DAG whose connected components are rooted directed trees. A component having one single node is called an *isolated node*. The *order* of a forest F, denoted by $|F|$, is its number of connected components. A tree is a forest of order one. All terms that are previously defined for trees can be naturally generalized to forests. In particular, a node v of a forest F is a leaf (resp. a root) if F has a component T such that v is a leaf (resp. the root) of T. If two nodes v and w are in two distinct components of F, the lowest common ancestor $LCA(v, w)$ does not exist.

1.2.2 Phylogenetic trees and forests

A full rooted binary tree is a rooted tree in which every internal node has exactly two children. Let T be a full rooted binary tree having $n := |V(T)|$ nodes. The following results are easy: (1) T has $\frac{n+1}{2}$ leaves and $\frac{n-1}{2}$ internal nodes; (2) the maximum distance between any two comparable nodes in T (this is also called the *diameter* of T) is bounded by $\frac{n-1}{2} = O(n)$.

An *ordered* full rooted binary tree is a full rooted tree that satisfies the following: for every internal node v, the set of children $ch(v)$ has a fixed ordering; we will write $ch(v)$ as an ordered pair instead of a set. The two children of v are respectively called the *left child* and the *right child* of v. When (v_1, v_2) are the children of v, the two subtrees $T|_{v_1}$ and $T|_{v_2}$ are called respectively the *left subtree* and the *right subtree* of T at node v.

Let X be a fixed arbitrary finite set, called the *set of taxa*. An X-tree is a tree whose leaves are bijectively labeled by the set X. The element of X assigned to a node v by the bijection is called the *label* of v.

Definition 1.1 (Phylogenetic trees and forests). Let X be a fixed set of taxa. A *phylogenetic tree* on X is a tree that satisfies all of the following properties: (1) it is a full rooted binary tree; (2) it is ordered; (3) it is an X-tree. A *phylogenetic forest* is a forest whose connected components are phylogenetic trees.

Biologically, two different phylogenetic trees according to Definition 1.1 can reflect the same evolutionary relationships between the taxa, because the ordering of the children of a node does not matter. We will use the notion of *isomorphic trees* to qualify phylogenetic trees that are biologically the same.

Definition 1.2 (Isomorphic phylogenetic forests). Two phylogenetic forests F_1 and F_2 on the same set X of taxa are *isomorphic*, denoted by $F_1 \cong F_2$,if there exists a bijection between their node sets $\varphi : V(F_1) \to V(F_2)$ that satisfies all of the following conditions:

1. For all pairs of nodes $(u, v) \in V^2(F_1)$, $(u, v) \in A(F_1) \iff (\varphi(u), \varphi(v)) \in A(F_2)$.
2. For each node $u \in V(F_1)$, u is a root of F_1 if and only if $\varphi(u)$ is a root of F_2.
3. For each leaf $u \in L(F_1)$, $\varphi(u)$ is a leaf of F_2, and the labels of u and $\varphi(u)$ are identical.

The first condition ensures that the two forests are isomorphic as directed graphs; the second condition additionally ensures that they are isomorphic as rooted forests, in other words, there exists a bijection between the components of two forests that identifies isomorphic rooted trees. Together, the first two conditions guarantee that the forests are the same as unordered forests, in other words, they have the same branching pattern, the same topology. The third condition ensures that the isomorphism pre-

serves the labels of the leaves. For two phylogenetic trees (i.e., forests of order one) T_1 and T_2 to be isomorphic, the second condition can be simplified to: $\varphi(r(T_1)) = r(T_2)$.

Remarks

For the sake of theoretical clarity, we have chosen a definition of the phylogenetic trees that is slightly different from the one that is commonly used in the literature; the latter does not consider the trees to be ordered. Our choice is motivated by the fact that, in many practical situations, it is more convenient or natural to consider ordered trees. In any such case, one must always bear in mind that two trees that "look different" may be biologically the same:

- Two phylogenetic trees that are biologically the same (isomorphic) can have different representations in Newick format, the standard format for representing trees in phylogenetic analysis (see Figure 1.3 for examples).

- For the purpose of visualizing phylogenetic trees on a two-dimensional graphic (we are doing a planar embedding), an ordering of the children of each node must be chosen. Visualization software such as [Yu+17] allow the user to flip the left and right subtrees manually. Two isomormphic trees can have different graphical representations (see Figure 1.3).

- When performing algorithms on phylogenetic trees, one often wants to traverse all the nodes in the tree. Assuming a sequential model of computation (i.e., not parallel), a convenient way of traversing an unordered tree would be to first choose a fixed ordering for the children of each node (so it becomes ordered), and then apply the standard traversal techniques for ordered binary trees (e.g., do a pre-order depth-first traversal: first the root, then the left subtree, finally the right subtree).

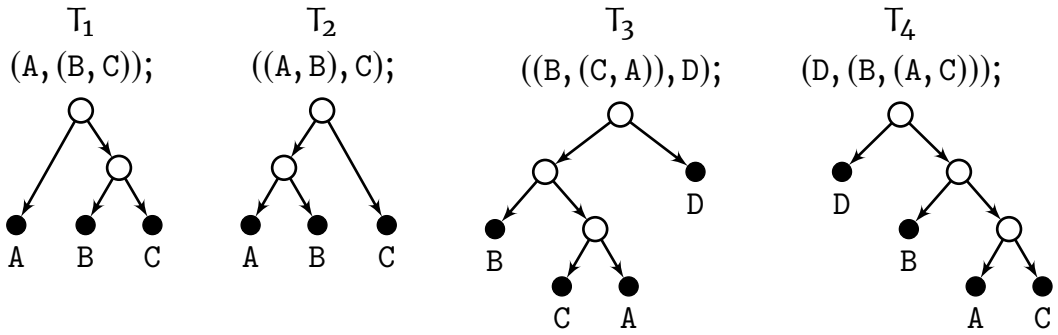


Figure 1.3: Four phylogenetic trees and their representations in Newick format. T_1 and T_2 are on the same set of taxa $\{A, B, C\}$ but are not isomorphic. T_3 and T_4 are on the same set of taxa $\{A, B, C, D\}$ and are isomorphic.

1.2.3 Concepts in algorithms and complexity

Dynamic programming

Dynamic programming (DP) is an easy and powerful method that is widely used for solving combinatorial optimization problems. First formally developed by Richard Bellman in the 1950's [Bel52; Bel13], it has found numerous applications in various fields. In bioinformatics, DP is used for solving problems such as Sequence alignment [NW70], Protein structure comparison [HS93], RNA structure prediction [RE99], etc.

In Chapter 4, we will present a formal model for characterizing problems that can be solved using DP or DP-like techniques. Here, we will only provide the necessary intuitions and present some terminologies that will be used in Chapter 2.

Simply put, dynamic programming is a general problem-solving method that tries to solve a problem by breaking it down into simpler *subproblems* in a recursive manner (DP-subproblem). To solve an optimization problem using a DP-algorithm, the value to be optimized should be computed via some recurrence relations using the optimal values of the subproblems. Usually, these optimal values of the subproblems are stored in a structure called the *dynamic programming table* (DP-table): one *cell* of the DP-table corresponds to a DP-subproblem and contains the optimal value of that subproblem. A typical DP algorithm has thus three components: (a) define the DP-subproblems (what does the DP-table look like?), (b) write down the recurrence relations (how to

fill up the DP-table?), (c) define the output (how to obtain the optimal value of the optimization problem from the DP-table?).

As an example, consider the following optimization problem, known as the 0-1 KNAPSACK PROBLEM: given a knapsack of capacity $W_{\max}$ and n items, each of value v_i and of weight w_i, the goal is to select a subset of items to put in the knapsack such that the total weight does not exceed $W_{\max}$ and the total value is maximized. In the case where the weights $w_1, \ldots, w_n$ and $W_{\max}$ are positive integers, there is a well-known DP-algorithm, which we will now describe by identifying the three components. (a) A subproblem, indexed by (i, j), asks to find the maximum value $M(i, j)$ that can be attained using the first i items and with total weight at most j, where $0 \leq i \leq n$ and $0 \leq j \leq W_{\max}$. (b) The recurrence relations are:

$$M(0, j) = 0,$$

$$M(i, j) = \begin{cases} M(i-1, j) & \text{if } j < w_i, \\ \max\{M(i-1, j),\ M(i-1, j-w_i) + v_i\} & \text{otherwise.} \end{cases}$$

(c) The output is $M(n, W_{\max})$.

A problem may be solved by several different DP-algorithms. Depending on the formulation, the time and space complexity of a DP-algorithm may or may not be a polynomial of the input size (the above algorithm for the 0-1 KNAPSACK PROBLEM has pseudopolynomial complexity).

Counting, enumeration

An optimization problem asks to find one solution, among all the feasible solutions, such that its value (or measure) maximizes or minimizes the objective function. For each optimization problem, there is a corresponding decision problem that asks whether there is a feasible solution for some particular value (a decision problem is simply a problem whose output set consists of only two values *yes* or *no*). This is called the *decision variant*, or the *associated* decision problem of the optimization problem. We will most often consider a NP optimization problem: every feasible solution has polynomial size, and its value can be computed in polynomial time. In this case, the associated decision problem will also be in the complexity class NP.

We can also define the *counting* and the *enumeration* problems that are associated with a given optimization problem. The first is concerned with establishing how many solutions are optimal. The second is concerned with enumerating, i.e., producing a list without duplicates of all optimal solutions. Counting and enumeration problems without an underlying optimization problem can be formulated in a similar manner.

If we have an algorithm for the enumeration problem, it can obviously be turned into an algorithm for the counting problem. However, in most cases, we do not have an upper bound for the number of optimal solutions that is polynomial in the size of the input. If we are interested in finding this number in polynomial time, the counting algorithm should not rely on enumeration. Just as for a decision problem in the class NP, we can either try to find a polynomial time counting algorithm, or try to show that it belongs to a subclass of "harder" problems (NP-complete for decision problems); the complexity class that characterizes the "harder" counting problems is the class #P-complete.

For the enumeration problem, since the output size can be exponential in the input size, the complexity is expressed in an output-sensitive manner. The performance of an enumeration algorithm can be measured in different ways:

- We can measure the running time in terms of the *total time* that is required for producing all solutions.
- By looking into the intermediate steps of the algorithm, we can measure the running time more carefully in terms of (1) the *preprocessing time*, i.e., the time required to produce the first solution, and (2) the maximum *time delay* between two consecutive outputs.

An algorithm is said to be *total polynomial time* or *output polynomial time* if its total running time is a polynomial of the input size and the output size. A total polynomial time algorithm is said to be *incremental polynomial time* if the time delay between the k-th and the $(k+1)$-th outputs is a polynomial of k and the input size. A total polynomial time algorithm is said to be *polynomial delay* if the time delay between two consecutive outputs is a polynomial of the input size. In Chapters 2 and 4, we will see several polynomial delay enumeration algorithms.

Approximation, heuristic

An approximation algorithm is an efficient algorithm (polynomial time) that finds an *approximate solution* to an optimization problem, with provable guarantees on the quality of the approximate solution: this is usually a multiplicative factor called the *approximation ratio*. For a minimization problem, a k-approximation algorithm, that is, an approximation algorithm with ratio k, guarantees to output a feasible solution whose value is within k times the optimal (minimum) value. Searching for approximation algorithms is a widely used approach for dealing with NP-hard optimization problems.

Some algorithms are known as *heuristics* as they provide an approximate solution efficiently but do not provide a guaranteed approximation ratio. Another type of algorithm, also called *heuristic*, provides exact solutions with no guaranteed running time bound. We will encounter both kinds of heuristics in this manuscript.

Chapter 2

Phylogenetic tree reconciliation: theoretical contributions

2.1 Introduction and definitions

2.1.1 The RECONCILIATION PROBLEM

In this section, we will define formally the PHYLOGENETIC TREE RECONCILIATION PROBLEM (shortly, the RECONCILIATION PROBLEM). At the end of the section, we will discuss the biological relevance of this definition and provide some references.

Reconciliation, events, event vector

Let H and P be two phylogenetic trees (see Definition 1.1), respectively on the set of taxa of host and parasite species. Let σ be a function from the parasite leaves $L(P)$ to the host leaves $L(H)$, representing the parasite/host associations between the taxa (i.e., the extant species). The triple (H, P, σ) will be called a *dataset*.

A *reconciliation* is simply a function ϕ that assigns, for each parasite node $p \in V(P)$, a host node $\phi(p) \in V(H)$. It should represent the parasite/host associations for the extant species as well as for all the ancestral parasite species in the tree. Naturally, the function ϕ must *extend* the leaf association function σ, that is, for each $p \in L(P)$, we

should have $\phi(p) = \sigma(p)$.

Now, for an internal node p, what are the constraints on the associated host $\phi(p)$? In the event-based model, a reconciliation must induce *events*. Concretely, for each internal node p *under a reconciliation* ϕ, the events at p will have two components: (1) exactly one event type (denoted by a special symbol) among $\mathbb{C}, \mathbb{D}, \mathbb{S}$, meaning respectively cospeciation, duplication, and host-switch, and (2) zero or more loss events (denoted by a nonnegative integer). The event type and the number of losses at a node p will depend on the *relationship between three host nodes* in the host tree: the host $\phi(p)$ assigned to p by ϕ, and the hosts $\phi(p_1)$ and $\phi(p_2)$ which are assigned to the two children (p_1, p_2) of p. Those dependencies can be expressed as an event type function E and a loss number function ξ, both taking as input three host nodes. The definitions of E and ξ are given in Table 2.1.

Case	**Value of** $E(h, h_1, h_2)$	**Value of** $\xi(h, h_1, h_2)$
Both h_1, h_2 are descendants of h, and $h_1 \not\prec h_2$ and $h = \mathit{LCA}(h_1, h_2)$.	$\mathbb{C}$	$d_H(h, h_1) + d_H(h, h_2) - 2$
Both h_1, h_2 are descendants of h, and the previous case does not apply.	$\mathbb{D}$	$d_H(h, h_1) + d_H(h, h_2)$
Either (1) h_1 is a descendant of h and $h_2 \not\prec h$, or (2) h_2 is a descendant of h and $h_1 \not\prec h$.	$\mathbb{S}$	Either (1) $d_H(h, h_1)$ or (2) $d_H(h, h_2)$
Any other case, that is, either (1) both $h_1 \not\prec h$ and $h_2 \not\prec h$, or (2) h is a proper descendant of h_1 or h_2.	nonvalid	nonvalid

Table 2.1: Definition of the event type function and the loss number function, both taking as input three host nodes.

Using the functions E and ξ, we now give in Definition 2.1 the formal definition of a reconciliation. In Definition 2.2, we also define the event type $E_\phi(p)$ and the loss number $\xi_\phi(p)$ of a parasite node p under a reconciliation ϕ, as these notions will be ubiquitous in the subsequent theoretical developments.

Definition 2.1 (Reconciliation). Given two phylogenetic trees H and P, and a function $\sigma\colon L(P) \to L(H)$, a function $\phi\colon V(P) \to V(H)$ is called a *reconciliation* of (H, P, σ) if it satisfies the following two conditions:

1. For every leaf node $p \in L(P)$,
$$\phi(p) = \sigma(p).$$
2. For every internal node $p \in V(P) \setminus L(P)$ with children (p_1, p_2):
$$E(\phi(p), \phi(p_1), \phi(p_2)) \in \{\mathbb{C}, \mathbb{D}, \mathbb{S}\}.$$

Definition 2.2 (Event type and loss number of a parasite node under a reconciliation). Let ϕ be a reconciliation of (H, P, σ). For a parasite node $p \in V(P)$, the *event type* $E_\phi(p)$ and the *loss contribution* $\xi_\phi(p)$ of p under ϕ are defined respectively as follows:

- If p is a leaf (here $\mathbb{T}$ is a special symbol for denoting the *terminal event*, i.e., the event type for the leaves):
$$E_\phi(p) := \mathbb{T}, \quad \xi_\phi(p) := 0.$$
- Otherwise, p is an internal node with children (p_1, p_2):
$$E_\phi(p) := E(\phi(p), \phi(p_1), \phi(p_2)), \quad \xi_\phi(p) := \xi(\phi(p), \phi(p_1), \phi(p_2)).$$

The image of the node-wise event type function E_ϕ is denoted by $\mathcal{E} := \{\mathbb{T}, \mathbb{C}, \mathbb{D}, \mathbb{S}\}$. The loss event is denoted by another special symbol $\mathbb{L}$. The values of E_ϕ partitions the set of internal parasite nodes into three disjoint subsets according to their event type; these subsets are denoted by $V^{\mathbb{C}}(P)$, $V^{\mathbb{D}}(P)$, $V^{\mathbb{S}}(P)$. The number of occurrences of each of the three event types together with the number of losses make up the *event vector* of the reconciliation ϕ, as defined in Definition 2.3. We also define in Definition 2.4 the *starred event vector* which is simply the event vector without the number of losses.

Definition 2.3 (Event vector). The *event vector* of a reconciliation ϕ is the quadruple of integers consisting of the total number of each type of events $\mathbb{C}$, $\mathbb{D}$, $\mathbb{S}$, $\mathbb{L}$, i.e.
$$\vec{e}(\phi) := \left(\left|V^{\mathbb{C}}(P)\right|, \left|V^{\mathbb{D}}(P)\right|, \left|V^{\mathbb{S}}(P)\right|, \sum_{p \in V(P)} \xi_\phi(p) \right). \tag{2.1}$$

Definition 2.4 (Starred event vector). The *starred event vector* of a reconciliation ϕ is the triple of integers that correspond to the first three elements of the event vector, i.e.,

$$\vec{e}^{\,*}(\phi) := \left(\left|V^{\mathbb{C}}(P)\right|, \left|V^{\mathbb{D}}(P)\right|, \left|V^{\mathbb{S}}(P)\right|\right). \tag{2.2}$$

The following result is trivial: the sum of the three elements of $\vec{e}^{\,*}(\phi)$ is equal to the number of internal parasite nodes, that is, $\frac{V(P)|-1}{2}$.

In Figure 2.1, for $1 \leq i \leq 6$, we show six different reconciliations ϕ_i, on the same dataset (H, P, σ). On the right side of each reconciliation ϕ_i, we also marked the event type $E_{\phi_i}(p)$ and the loss number $\xi_{\phi_i}(p)$ for each internal p, and the event vector $\vec{e}\,(\phi_i)$.

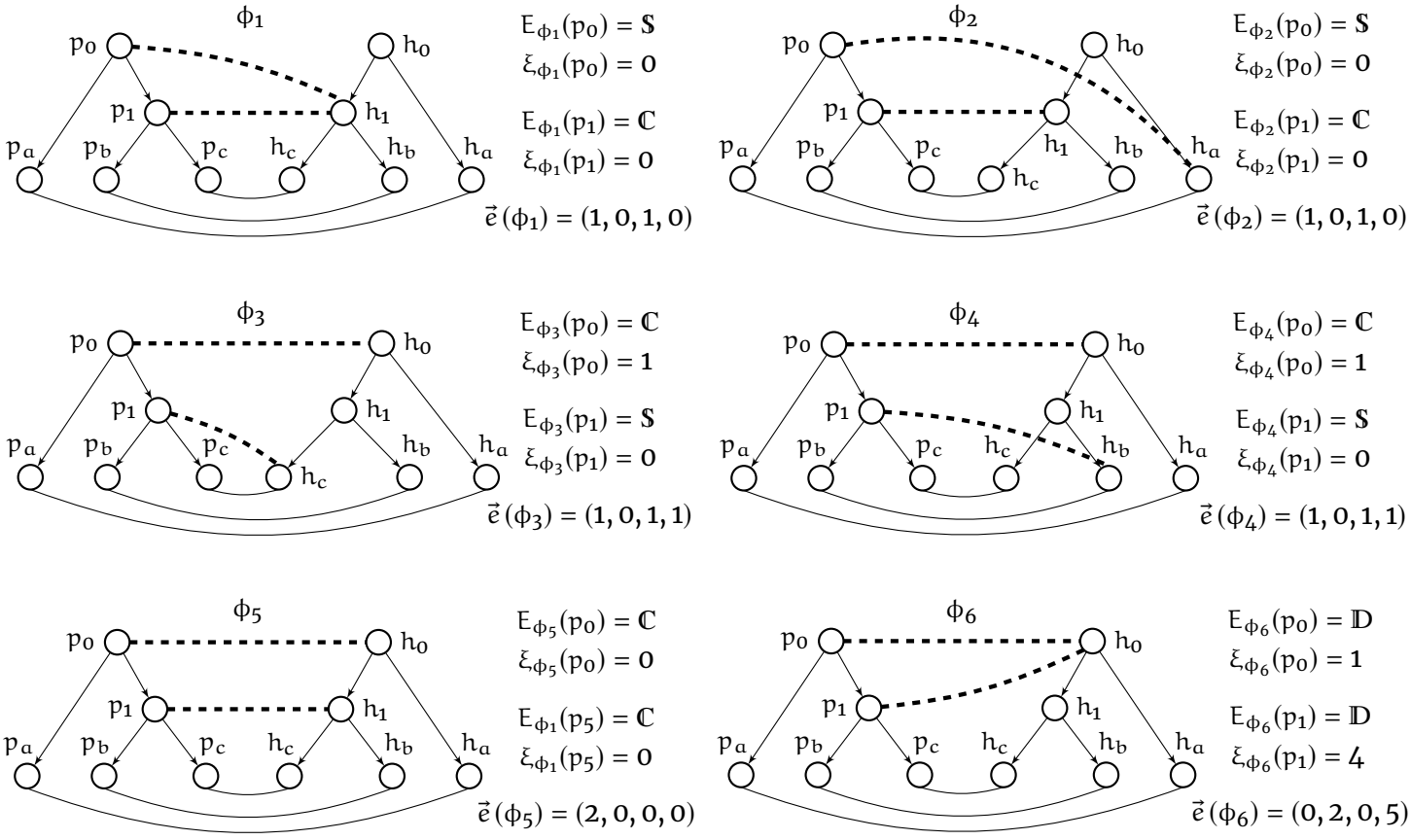


Figure 2.1: Example of six different reconciliations on the same dataset. For each reconciliation, the parasite and the host trees are drawn side by side. The solid curves at the bottom indicate the parasite/host associations for the parasite leaves; the dashed lines indicate the associations for the internal parasite nodes.

Cost of a reconciliation, RECONCILIATION PROBLEM

A *cost vector* $\vec{c} := (c(\mathbb{C}), c(\mathbb{D}), c(\mathbb{S}), c(\mathbb{L}))$ is a quadruple of real numbers representing the costs that we want to assign to each event type (including the loss event). Given a cost vector $\vec{c}$, the *cost of a reconciliation* ϕ is defined as the dot product between its event vector and the cost vector: $\vec{e}(\phi) \cdot \vec{c}$.

We are now ready to define the combinatorial optimization problem which we call the RECONCILIATION PROBLEM. Recall that a *dataset* is a triple (H, P, σ). An *instance* is a pair consisting of a dataset and a cost vector $((H, P, \sigma), \vec{c})$. Given an instance, the set of feasible solutions is the set of reconciliations of the dataset (H, P, σ). The goal of the optimization problem is to find an optimal feasible solution, that is, a reconciliation ϕ that minimizes the cost $\vec{e}(\phi) \cdot \vec{c}$. In the associated enumeration problem, the goal is to find all reconciliations of minimum cost.

For example, for the dataset depicted in Figure 2.1, if the cost vector is $(0, 0, 0, 0)$, all reconciliations are optimal; if the cost vector is $(0, 1, 1, 1)$, only ϕ_5 is optimal.

Remarks

The RECONCILIATION PROBLEM is studied extensively in the literature [THL11; BAK12; Don+15; Ma+18]. Notice that, for the sake of clarity, we used a slightly different set of notations to define the event-based model (in particular, we rely on the functions E and ξ in Table 2.1). The computational problem is equivalent to the ones that can be found in those references.

There exists in the literature a non-equivalent formulation of the RECONCILIATION PROBLEM which considers the host tree to be *dated*: the input also includes a time function that maps each host node to a positive number [Doy+10; Sco+13]. This different computational problem is *not* studied in the present manuscript.

For any given dataset, the set of feasible solutions is nonempty: a function ϕ that satisfies the leaf association constraint (condition 1. of Definition 2.1) and that maps every internal parasite node to the root of the host tree (for example, ϕ_6 of Figure 2.1) is always a valid reconciliation.

2.1.2 Time-feasibility

A phylogenetic tree T, which represents a possible scenario for the evolutionary history of species, implies the existence of a node ordering that is consistent with the ancestor/descendant relations in the tree. In practice, we can consider a time function $\tau\colon V(T) \to \mathbb{R}_{>0}$ that maps each node to a positive number and satisfies the following: for each arc $(v, w) \in A(T)$, we have $\tau(v) < \tau(w)$. By transitivity, if v is an ancestor of w, then $\tau(v) < \tau(w)$.

For a dataset (H, P, σ) of the RECONCILIATION PROBLEM, let τ_H be the time function for the host tree. For a reconciliation ϕ to be biologically meaningful, the following property must be satisfied:

$$\text{for each arc } (p, p_i) \in A(P), \quad \text{we should have } \tau_H(\phi(p)) < \tau_H(\phi(p_i)). \tag{2.3}$$

From the definition of a reconciliation (Table 2.1 and Definition 2.1), we can see that this desired property is already ensured (by noticing that $\phi(p)$ is an ancestor of $\phi(p_i)$), *except* for the arcs (p, p_i) involved in a host-switch event, that we define next:

Definition 2.5 (Host-switch arc). Let ϕ be a reconciliation of (H, P, σ). Let $p \in V(P)$ be an internal parasite node of event type $\mathbb{S}$ under ϕ, i.e., $E_\phi(p) = \mathbb{S}$, then the arc $(p, p_i) \in A(P)$ is called the *host-switch arc* of p under ϕ, where p_i is the unique child of p in P such that $\phi(p_i) \nsucc \phi(p)$.

The presence of the host-switch arcs can introduce incompatibilities in time, and as a result, a function τ_H satisfying Equation (2.3) can no longer exist. Such a reconciliation, while being a feasible reconciliation in our model, is called *time-unfeasible* or *cyclic* (examples can be found, for instance, in [THL11]). There are two important observations with respect to this concept: (1) given a reconciliation ϕ, whether ϕ is time-feasible can be decided in polynomial time [THL11]; (2) for any dataset (H, P, σ), the set of time-feasible reconciliations is nonempty (the reconciliation that maps every internal parasite node to the root of the host tree does not have any host-switches).

In order to only obtain biologically meaningful reconciliations, we can modify our RECONCILIATION PROBLEM into TIME-FEASIBLE RECONCILIATION PROBLEM: given an instance, an algorithm for the latter should output a reconciliation that minimizes

the cost among all time-feasible reconciliations. Unfortunately, TIME-FEASIBLE RECONCILIATION PROBLEM is NP-hard [Ova+11; THL11]. For the remainder of this chapter, we will concentrate on the RECONCILIATION PROBLEM, while keeping in mind that the solutions that we will explore may or may not be time-feasible. In Chapter 3, Section 3.1, we will see that, for many instances, by simply enumerating the optimal reconciliations and filtering out those that are not time-feasible, we can already obtain time-feasible reconciliations. We will also present in Chapter 3 some more methods for obtaining time-feasible reconciliations.

2.1.3 The reconciliation graph

Putting aside the time-feasibility and returning to our RECONCILIATION PROBLEM, we are interested in finding reconciliations of minimum cost. It turns out that this problem can be solved efficiently using the dynamic programming technique [BAK12; Don+15]. The DP-algorithm also solves the enumeration version of the problem: in fact, it produces a graph structure which is a compact representation of *all* reconciliations of minimum cost [Don+15; Ma+18]. One can argue that the power of this model (or the reason for its popularity) comes primarily from the fact that, even though the number of optimal solutions can grow exponentially with the size of the input trees, many features of the solution space can be effectively understood by exploring this graph structure of polynomial size [Ma+18]. In this section, we will describe the DP-algorithm for constructing the graph that we call the *reconciliation graph*. This graph will serve as the basis for a number of problems that we will tackle later in the chapter.

Dynamic programming

We will first describe a DP-algorithm that finds the *optimal value*, i.e., the minimum cost of any reconciliation. Using the terminologies from Section 1.2.3, given an instance $((H, P, \sigma), \vec{c})$, we need to: (a) define the DP-subproblems and the DP-table, (b) write down the recurrence relations for filling up the DP-table, (c) define the output.

Recall that $\mathcal{E} := \{\mathbb{T}, \mathbb{C}, \mathbb{D}, \mathbb{S}\}$ is the set of possible event types for a node. Given a parasite node $p \in V(P)$, we denote by $P|_p$ the subtree of P rooted at node p, and by $\sigma|_{L(P|_p)}$ the

restriction of σ to the leaves of $P|_p$. For the purpose of describing the DP-subproblems, for a fixed dataset (H, P, σ), we will take the subtree of P at a particular node $p \in V(P)$. In this context, a reconciliation of the dataset $(H, P|_p, \sigma|_{L(P|_p)})$ is simply called a *reconciliation of* $P|_p$.

The set $U := V(P) \times V(H) \times \mathcal{E}$ is called the *space of cells* of the DP-table. A triple $(p, h, e) \in U$ is called a *cell* of the DP-table. A *DP-subproblem*, indexed by a cell (p, h, e), asks to find a reconciliation ϕ of $P|_p$ that minimizes the cost, and satisfies $\phi(p) = h$ and $E_\phi(p) = e$. The DP-table stores at each cell the optimal value of the subproblem, i.e., the minimum cost, denoted by $M(p, h, e)$. The recurrence relations for computing $M(p, h, e)$ are as follows:

- If p is a leaf,

$$M(p, h, e) = \begin{cases} 0 & \text{if } h = \sigma(p) \text{ and } e = \mathbb{T}, \\ +\infty & \text{otherwise.} \end{cases} \tag{2.4}$$

- Otherwise, p is an internal node with children (p_1, p_2). In this case,

$$M(p, h, e) = \min_{\substack{E(h,h_1,h_2)=e \\ h_1,h_2 \in V(H) \\ e_1,e_2 \in \mathcal{E}}} M(p_1, h_1, e_1) + M(p_2, h_2, e_2) + c(e) + c(\mathbb{L})\, \xi(h, h_1, h_2). \tag{2.5}$$

The output, i.e., the minimum cost of any reconciliation of (H, P, σ) is given by

$$\min_{h \in V(H), e \in \mathcal{E}} M(r(P), h, e). \tag{2.6}$$

ad-AND/OR graphs

Before describing how to construct the reconciliation graph, we will first define a more general graph structure: the *acyclic decomposable AND-OR graph* (ad-AND/OR graph), which is known for having an intimate relationship with dynamic programming on a tree. The reconciliation graph will be a particular ad-AND/OR graph with some additional properties. In Chapter 4, we will study the relationship between general DP-algorithms and ad-AND/OR graphs and provide some references.

Definition 2.6 (ad-AND/OR graph). A directed graph G is an *acyclic decomposable AND/OR graph* (shortly, ad-AND/OR graph) if it satisfies the following:

- G is a acyclic (it is a DAG).
- G is bipartite: its node set $V(G)$ can be partitioned into $(\mathcal{A}, \mathcal{O})$ so that all arcs of G are between these two sets. Nodes in $\mathcal{A}$ are called *AND nodes*; nodes in $\mathcal{O}$ are called *OR$^+$ nodes*.
- Every AND node has in-degree at least one and out-degree at least one. The set of nodes of out-degree zero is then a subset of $\mathcal{O}$ and is called the set of *goal nodes*; the remaining OR$^+$ nodes are simply the *OR nodes*. The subset of OR nodes of in-degree zero is the set of *start nodes*.
- G is *decomposable*: for any AND node, the sets of nodes that are reachable from each one of its child nodes are pairwise disjoint.

An ad-AND/OR graph represents the *space of solutions* for some problem instance. A solution will be a particular kind of subgraph which is a rooted tree.

Definition 2.7 (Solution subtree)**.** A *solution subtree* T of an ad-AND/OR graph G is a subgraph of G which: (1) contains exactly one start node; (2) for any OR node in T it contains one of its child nodes in G, and for any AND node in T it contains all its children in G.

The set of solution subtrees of G is denoted by $\mathcal{T}(G)$. It is immediate to see that a solution subtree is indeed a subtree of G: it is a rooted tree, the root of which is a start node. If we would drop the requirement of G being decomposable, the object defined in Definition 2.7 would not be guaranteed to be a tree.

It is easy to see that the following four-step procedure allows to obtain one solution subtree: (1) start at any start node, (2) for any visited OR node, visit one child, (3) for any visited AND node, visit all children, (4) stop when the goal nodes are reached. In fact, a well-known folklore result states that the set $\mathcal{T}(G)$ of solution subtrees of any ad-AND/OR graph G can be enumerated efficiently: after a pre-processing step in time linear in the size of G, the delay between outputting two consecutive solutions is linear in the size of the solution. In Chapter 4 we will discuss this result again.

We will define now a notion of subgraph for ad-AND/OR graphs that we will use extensively later in this chapter as well as in Chapter 4. Intuitively, the relationship

between such a subgraph and the entire graph correspond to the relationship between a DP-subproblem and the "full" optimization problem.

Definition 2.8 (Subgraph starting from a set of nodes). Let G be an ad-AND/OR graph. Let $\mathcal{O}$ be a set of OR$^+$ nodes of G. The *subgraph of* G *starting from* $\mathcal{O}$, denoted by $G/\mathcal{O}$, is the subgraph obtained from G by setting $\mathcal{O}$ as the new set of start nodes (i.e., by removing all nodes are *unreachable* from $\mathcal{O}$ through directed paths).

Constructing the reconciliation graph, first properties

We can finally describe now how to construct the *reconciliation graph* of a given instance of the RECONCILIATION PROBLEM from Equations (2.4) and (2.5), the recurrence relations for filling up the DP-table.

The construction is done in two steps. In the first step, we build a directed graph in which every node retains an additional attribute, its *value*, and every OR$^+$ node is uniquely *labeled by* a cell $(p, h, e) \in U$ of the DP table. In the second step, we *prune* the graph by removing nodes that do not yield optimal values.

1. For each $(p, h, e) \in U$ such that p is a leaf, create a goal node labeled by (p, h, e); its value is equal to 0 if $h = \sigma(p)$ and $+\infty$ otherwise (see Equation (2.4)). Then, for each $(p, h, e) \in U$ in the post-order of $V(P) \setminus L(P)$, let (p_1, p_2) be the children of p,
   i. For each (p_1, h_1, e_1) and each (p_2, h_2, e_2) such that $E(h, h_1, h_2) = e$, create an AND node, connect it to the two OR$^+$ nodes respectively labeled by (p_1, h_1, e_1) and (p_2, h_2, e_2). Its value is equal to the sum of the values of its two children, plus $c(e) + c(\mathbb{L})\, \xi(h, h_1, h_2)$ (see Equation (2.5)).
   ii. Create a single OR node, connect it to every AND node created in the previous step. Its label is (p, h, e), and its value is the minimum of the values of its children.
2. For each $(r(P), h, e) \in U$, remove the OR node labeled by that cell unless its value is equal to the optimal cost (see Equation (2.6)). For each OR node w, remove the arc to its child AND node w_i if the value of w_i is not equal to the value of w. Finally, remove recursively all AND nodes without incoming arcs.

It can be checked that the reconciliation graph is indeed an an-AND/OR graph as defined in Definition 2.6. An OR$^+$ node labeled by (p, h, e) is a start node if and only if $p = r(P)$, and is a goal node if and only if $p \in L(P)$. It is also immediate to see that each AND node in the reconciliation graph has exactly one in-neighbor and exactly two children.

We will consider the two children of each AND node to be *ordered*. More precisely, for an AND node v, if its in-neighbor is labeled by (p, h, e) where p has two ordered children (p_1, p_2), we know that the two children of v are respectively labeled by cells of the form (p_1, h_1, e_1) and (p_2, h_2, e_2): we will call the first one the *left child* w_1 and the second one the *right child* w_2 of v, and we will write $\boldsymbol{ch}(v) = (w_1, w_2)$. However, the set of children of an OR node w is *unorderd*, and we will write $\boldsymbol{ch}(w) = \{v_i\}$.

For an OR node, we will typically be interested not in its set of children but in its set of "grandchildren", hence, we introduce here a new notation. If w is an OR node, we call the *grandchild couples* of w, denoted by $\boldsymbol{gch}(w)$, the union of the children of its child AND nodes (it is a set of couples of OR$^+$ nodes): $\boldsymbol{gch}(w) := \bigcup_{v_i \in \boldsymbol{ch}(w)} \boldsymbol{ch}(v_i)$. By construction, no two AND nodes that are children of the same OR node have the same couple of children, therefore, the number of grandchild couples is equal to the number of child AND nodes: $|\boldsymbol{gch}(w)| = |\boldsymbol{ch}(w)|$. Notice that an OR$^+$ node can appear as a grandchild of two different nodes, and can also appear in two different grandchild couples of a same node (see Figure 2.2).

Let $n = |V(H)|$ and $m = |V(P)|$. It is clear that the reconciliation graph has $O(mn)$ OR$^+$ nodes. Each OR node has $O(n^2)$ children (or, equivalently, $O(n^2)$ grandchild couples). The space required for storing the reconciliation graph is thus $O(mn^3)$. Practical algorithms *do not* construct the reconciliation graph in the same way as we just described, as the functions Σ and ξ are not trivial to compute. Nevertheless, the reconciliation graph can be constructed using $O(mn^3)$ time and space complexity [Don+15].

Solution subtrees of the reconciliation graph are optimal reconciliations

As announced in the beginning of this section, the reconciliation graph is a compact representation of the set of optimal reconciliations. Formally, if G is the reconcilia-

tion graph for a given instance of the RECONCILIATION PROBLEM, then the set $\mathcal{T}(G)$ of solution subtrees of G corresponds bijectively to the set of optimal solutions of that instance, i.e., the set of reconciliations of minimum cost. Moreover, we have the following result for the subgraph $G/\{w\}$ starting from an OR$^+$ node w labeled by (p, h, e): the set $\mathcal{T}(G/\{w\})$ of solution subtrees of the subgraph corresponds bijectively to the set of optimal solution of the DP subproblem indexed by (p, h, e).

In practice, to convert a solution subtree $T_1 \in \mathcal{T}(G)$ into a reconciliation ϕ, we only need to look at the labels (p, h, e) of the OR$^+$ nodes in T_1 (notice that a reconciliation can simply be viewed as a collection of triples of the form (p, h, e)). We will henceforth use interchangeably the terms *solution subtrees* of the reconciliation graph and *optimal reconciliations* of the problem instance.

Let $n = |V(H)|$ and $m = |V(P)|$. As we mentioned, the reconciliation graph can be constructed in $O(mn^3)$ time. After the construction, using the standard techniques for ad-AND/OR graphs, the total number of optimal reconciliations can be easily computed, and the solutions subtrees can be enumerated with a linear time delay, that is $O(m)$. Therefore, there is an algorithm with a $O(mn^3)$ time pre-processing step and $O(m)$ time delay for enumerating all optimal reconciliations.

Figure 2.2 shows a reconciliation graph based on the same dataset (H, P, σ) as in Figure 2.1 with nine solution subtrees. Among these nine reconciliations, four have event vector $(0, 0, 2, 0)$, two have $(1, 0, 1, 0)$ (ϕ_1 and ϕ_2 of Figure 2.1), two have $(1, 0, 1, 1)$ (ϕ_3 and ϕ_4 of Figure 2.1), and one has $(2, 0, 0, 0)$ (ϕ_5 of Figure 2.1). The reconciliation shown in bold is ϕ_4 of Figure 2.1.

2.2 Defining equivalence relations

In this section, we will start by giving the formal definitions for a number of different equivalence relations on the set of reconciliations of a given dataset, then we will explain the motivation for such definitions, show some preliminary results, and state the computational problems that are related to those equivalence relations.

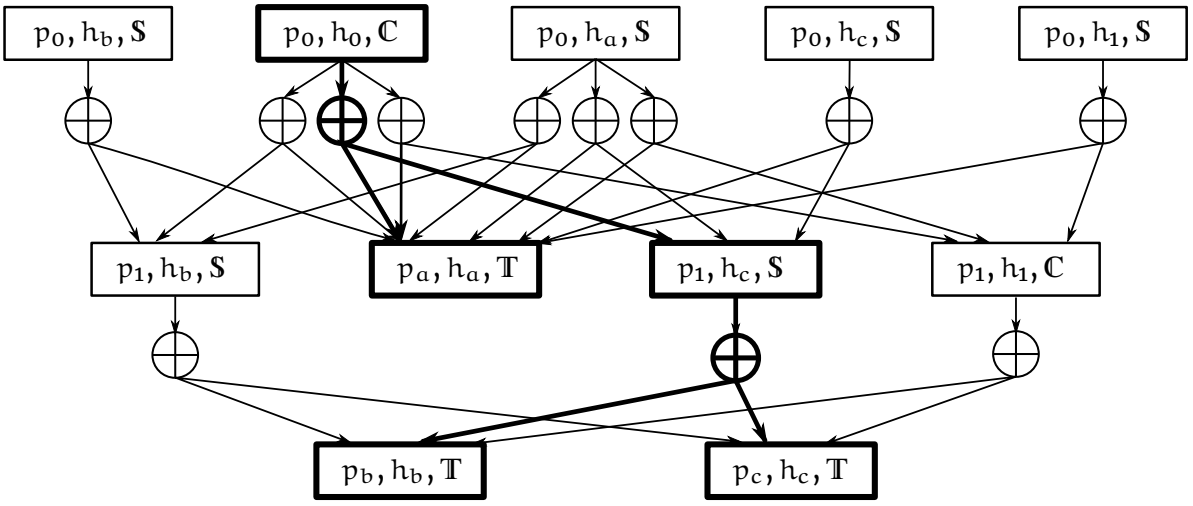


Figure 2.2: Example of a reconciliation graph for the dataset (H, P, σ) of Figure 2.1. Crossed circles are AND nodes. Rectangles are OR^+ nodes. The cells with which the OR^+ nodes are labeled are written inside. One solution subtree is shown in bold (it is ϕ_4 of Figure 2.1).

2.2.1 Definitions

Definition 2.9 (V-equivalence). Two reconciliations ϕ_1 and ϕ_2 are *Vector equivalent* (shortly, *V-equivalent*), if their event vectors are the same: $\vec{e}\,(\phi_1) = \vec{e}\,(\phi_2)$.

Definition 2.10 (V^*-equivalence). Two reconciliations ϕ_1 and ϕ_2 of the same dataset are *Starred Vector equivalent* (shortly, *V^*-equivalent*), if their starred event vectors are the same: $\vec{e}^{\,*}(\phi_1) = \vec{e}^{\,*}(\phi_2)$.

Definition 2.11 (E-equivalence). Two reconciliations ϕ_1 and ϕ_2 of the same dataset are *Event equivalent* (shortly, *E-equivalent*), if $E_{\phi_1}(p) = E_{\phi_2}(p)$ for all $p \in V(P)$.

Definition 2.12 (EL-equivalence). Two reconciliations ϕ_1 and ϕ_2 of the same dataset are *Event Labeling equivalent* (shortly, *EL-equivalent*), if (1) they are E-equivalent, and (2) the host-switch arcs are the same: for all p such that $E_{\phi_1}(p) = \mathbb{S}$, the host-switch arcs at p under ϕ_1 and under ϕ_2 (see Definition 2.5) are the same.

Definition 2.13 (CD-equivalence). Two reconciliations ϕ_1 and ϕ_2 of the same dataset are *Cospeciation Duplication equivalent* (shortly, *CD-equivalent*), if (1) they are E-equivalent, and (2) for all p such that $E_{\phi_1}(p) \in \{\mathbb{C}, \mathbb{D}\}$, we have $\phi_1(p) = \phi_2(p)$.

For example, ϕ_1 and ϕ_2 (or ϕ_3 and ϕ_4) of Figure 2.1 are V-, V*-, E-, and CD-equivalent but not EL-equivalent; ϕ_1 and ϕ_3 are V*-equivalent, but they are not equivalent under any other equivalence relation.

Each one of these equivalence relation splits the set of optimal reconciliations of a given instance into *equivalence classes*, i.e., subsets of pairwise equivalent reconciliations. One *representative* of an equivalence class is simply a reconciliation in the corresponding subset. We will abuse the terminology and call equivalence classes the objects that best represent the common property of the reconciliations in that subset (e.g., a V-equivalence class is simply a vector). A reconciliation *in an equivalence class* will then be a reconciliation satisfying that property.

We will now introduce the notations for denoting the equivalence classes and making statements such as "a reconciliation is in an equivalence class". Recall that, if G is the reconciliation for a given instance, then $\mathcal{T}(G)$ denotes the set of optimal reconciliations. For a fixed equivalence relation, we consider a function π that maps each reconciliation $\phi \in \mathcal{T}(G)$ to the equivalence class $\pi(\phi)$ of ϕ (in other words, ϕ is in the equivalence class $\pi(\phi)$). For the E-, EL-, and CD-equivalence relations, the "common property" of a class will be expressed via some local properties on each parasite node; we define in Definition 2.14 this localized property that we call the *color* of a node. Then, the definitions of π for each equivalence relation are given in Definition 2.15.

Definition 2.14 (Color of a parasite node under a reconciliation). Let ϕ be a reconciliation of (H, P, σ). For each parasite node $p \in V(P)$, the *color* $\kappa_\phi(p)$ of p under ϕ is defined in Table 2.2.

Equivalence relation	**The color $\kappa_\phi(p)$ of p under ϕ** (it is either a couple or a triple)
E-equivalence relation	$(p, E_\phi(p))$
EL-equivalence relation	$\begin{cases} (p, E_\phi(p)) & \text{if } E_\phi(p) \neq \$ \\ (p, \$_L) & \text{if } (p, p_1) \text{ is the host-switch arc at } p \\ (p, \$_R) & \text{otherwise, } (p, p_2) \text{ is the host-switch arc} \end{cases}$ where $\$_L$ and $\$_R$ are two special symbols
CD-equivalence relation	$\begin{cases} (p, \phi(p), E_\phi(p)) & \text{if } E_\phi(p) \neq \$ \\ (p, ?, \$) & \text{otherwise, } E_\phi(p) = \$ \end{cases}$ where ? is a special symbol (denoting unknown host)

Table 2.2: For each of E-, EL-, and CD-equivalence relations, definition of the color of a parasite node under a reconciliation.

Definition 2.15 (Equivalence class). Consider a fixed instance on a dataset (H, P, σ) and denote by $\mathcal{T}(G)$ the set of optimal reconciliations. The *equivalence class* $\pi(\phi)$ of a reconciliation ϕ is the object defined in Table 2.3.

Equivalence relation	**The equivalence class $\pi(\phi)$ of ϕ**
V-equivalence relation	$\vec{e}\,(\phi)$ (it is a vector of four integers)
V*-equivalence relation	$\vec{e}^{\,*}(\phi)$ (it is a vector of three integers)
E-, EL-, CD-equivalence relations	$\pi(\phi)$ is a function that maps each $p \in V(P)$ to its color $\kappa_\phi(p)$

Table 2.3: For each equivalence relation, definition of the equivalence class of a reconciliation.

For a fixed equivalence relation, the *set of equivalence classes* of the instance, denote by $\pi(\mathcal{T}(G))$, is defined as follows:

$$\pi(\mathcal{T}(G)) := \bigcup_{\phi \in \mathcal{T}(G)} \pi(\phi). \tag{2.7}$$

2.2.2 Motivations

The first and foremost motivation of defining equivalence relations is the need of capturing useful biological information from the set of optimal reconciliations, when this set is too large for manual analyses or for exhaustive enumeration. While the number of optimal reconciliations can grow exponentially fast with the size of the input trees, we will see in Section 2.3.1 that the number of V- or V^*-equivalence classes has a polynomial upper bound. The set of V-equivalence classes (i.e., the event vectors) provides already a first information about the co-evolutionary history of the hosts and their parasites. Indeed, a high number of cospeciations may indicate that hosts and parasites evolved together, while a high number of host-switches may indicate that the parasites are able to infect different host species. Under the scope of the E-equivalence relation, we are also interested in which parasites are associated to each event type (disregarding losses): by looking at the E-equivalence classes, we can get knowledge of the number as well as the locality of the events inside the parasite tree. The information from the E-equivalence classes can be further refined using the EL-equivalence classes, as they tell us not only where the host-switches happen but also which arcs of the parasite tree are involved (this information is relevant in the analysis of time-feasibility; a practical application can be found in Chapter 3, Section 3.1.4).

The intuition behind the CD-equivalence relation is that, when a host-switch happens, there may be various hosts that can be selected as the parasite's "landing site". In this case, we choose to consider as equivalent those reconciliations for which, while the hosts that receive the switching parasites may differ, all the other parasite/host associations (not corresponding to a host-switch) are the same. These reconciliations are similar and often indistinguishable without additional biological information. Indeed, take the two reconciliations ϕ_1 and ϕ_2 in Figure 2.1: they are identical except for one switching parasite p_0, which is mapped to h_1 by ϕ_1 and to h_a by ϕ_2. Since h_1 and h_a are two sibling nodes sharing the same parent in the host tree, without further information, there is no good way to tell apart the two reconciliations ϕ_1 and ϕ_2, hence we consider them as equivalent.

Among the five equivalence relations, the CD-equivalence relation is the only one

that captures the precise parasite/host associations for some internal parasite nodes. At first glance, it might seem too restrictive, and one might fear that the number of CD-equivalence classes would be too large, which limits the practical usability. In Chapter 3, Section 3.1, we will see that, for real-world datasets, the number of CD-equivalence classes is almost always much smaller compared to the number of optimal reconciliations. In fact, there is also a theoretical foundation for allowing the host-switch associations to be "movable": as we will see in Lemma 2.1, finding one optimal reconciliation means that (a potentially very large number of) certain other reconciliations must also be optimal, all of them differ only by the associations of the host-switch nodes.

Lemma 2.1. *(The proof is straightforward and can be found in my Master thesis). Let* ϕ *be an optimal reconciliation of* (H, P, σ) *with a cost vector* $\vec{c}$ *satisfying* $c(\mathbb{C}) < c(\mathbb{D}) + 2\,c(\mathbb{L})$ *(this is verified by any reasonable cost vector used in practice). Let* $p_0 \in V(P)$ *be an internal node with children* (p_1, p_2)*. Suppose* $E_\phi(p_0) = \$$ *and suppose wlog that* (p_0, p_1) *is the host-switch arc at* p_0*. Suppose that* $\phi(p_0) \neq \phi(p_2)$ *(so necessarily* $\phi(p_0)$ *is a proper ancestor of* $\phi(p_2)$*, see Figure 2.3). Take* ***any*** *node* h *on the path from* $\phi(p_0)$ *to* $\phi(p_2)$ *in* H*,* ***excluding*** $\phi(p_0)$ *(there is at least one choice for* h *which is* $\phi(p_2)$*). Consider* $\phi' : V(P) \to V(H)$ *such that*

$$\phi'(p_0) = h, \quad \textit{and } \forall p \in V(P) \setminus \{p_0\},\ \phi'(p) = \phi(p)$$

(ϕ' *is identical to* ϕ *except for the association at* p_0*). Then* ϕ' *is an optimal reconciliation and is CD-equivalent to* ϕ*.*

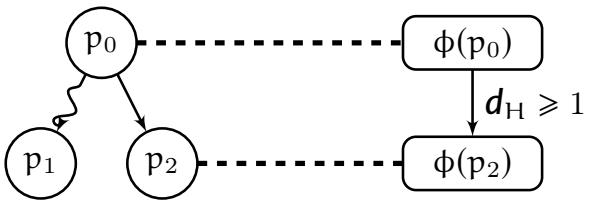


Figure 2.3: Illustration of Lemma 2.1. Circles are parasite nodes; rectangles with rounded corners are host nodes; dashed lines represent the parasite/host associations; the zigzag line indicates that (p_0, p_1) is the host-switch arc at p_0 under ϕ.

2.2.3 Relationship between the equivalence relations

From the Definitions 2.9–2.13, we can easily deduce the following:

1. If ϕ_1 and ϕ_2 are V-equivalent, then they are V^*-equivalent.
2. If ϕ_1 and ϕ_2 are E-equivalent, then they are V^*-equivalent.
3. If ϕ_1 and ϕ_2 are EL-equivalent, then they are E-equivalent.
4. If ϕ_1 and ϕ_2 are CD-equivalent, then they are E-equivalent.

The next result is only guaranteed when ϕ_1 and ϕ_2 are optimal reconciliations for a cost vector $\vec{c}$ satisfying $c(\mathbb{L}) \neq 0$ (nonzero loss cost):

5. If ϕ_1 and ϕ_2 are V^*-equivalent, then they are V-equivalent.

The converse of the above five statements is generally *not true*; counterexamples can be found easily.

In practice, we will often look at the number of equivalence classes of a fixed instance (i.e., the size $|\pi(\mathcal{T}(G))|$ of the set of equivalence classes, see Definition 2.15) for several different equivalence relations. If we know that being A-equivalent implies being B-equivalent, where A and B are two equivalence relations, then we know that the number of A-equivalence classes is no bigger than the number of B-equivalence classes. If the experimental observation contradicts this, there is a strong indication that something went wrong (for example, we did not compute the things that we thought we did). Therefore, knowing the relationship between the equivalence relations, apart from the obvious usefulness of helping us to better understand the meaning of equivalence classes, is also useful during experiments as a safeguard measure.

2.2.4 Computational problems

Equipped with our definitions of equivalence classes (see Definition 2.15), we aim at studying the features of the set of optimal reconciliations by enumerating the equivalence classes (see Chapter 3 for some other approaches, and a comparison with the equivalence-based approach). Naively, one would iterate through every reconciliation

and record their properties, then report the equivalence classes, and, only at the end, report the statistics of the reconciliations in each equivalence class. However, when the number of reconciliations is too large, for example, in the order of 10^{42} (see Chapter 3, Section 3.1.2 and [Wan+20]), the naive method is not feasible.

The challenge is then to enumerate *directly* the equivalence classes *without* enumerating the optimal reconciliations explicitly. Concretely, the set of optimal reconciliations will be represented implicitly as $\mathcal{T}(G)$, the set of solution subtrees of a reconciliation graph G. Given a reconciliation graph as input, we will tackle the following problems:

- Count the number of equivalence classes.
- Enumerate the equivalence classes.
- Study a particular equivalence class. That is, given an equivalence class,
  - Count the number of reconciliations in that class,
  - Find one representative (i.e., one optimal reconciliation) of that class,
  - Enumerate all reconciliations in that class.

2.3 Dealing with the V- and V*-equivalence relations

2.3.1 V-equivalence relation

The enumeration of the V-equivalence classes (i.e., event vectors) can be achieved by modifying the dynamic programming algorithm in such a way that the set of V-equivalence classes for each DP subproblem is kept along the way. This basic idea can, in theory, be applied to any equivalence relation. However, such a modified algorithm will require explicitly storing all the equivalence classes. Fortunately, for the V-equivalence relation, the space complexity will remain a polynomial of the size of the input trees, as we will see next.

Let $n = |V(H)|$ and $m = |V(P)|$. The first three elements of any event vector necessarily sum up to $\frac{m-1}{2}$, the number of internal parasite nodes, hence there are only $O(m^2)$

possible combinations. The number of losses $\xi_\phi(p)$ for each parasite node p under any reconciliation ϕ is at most twice the diameter of the host tree (i.e., twice the maximum distance between two nodes), so the fourth element of any event vector is bounded by $O(nm)$. Therefore, the number of event vectors is bounded by $O(nm^3)$.

Given an instance and its reconciliation graph G, we are interested in the following two problems: listing all event vectors, and, given a particular event vector, listing one (or all) optimal reconciliations in that V-equivalence class. We will now describe how to modify the DP algorithm for constructing the reconciliation graph G to solve both problems.

Recall that if w is an OR$^+$ node of the reconciliation graph G, the set of solution subtrees $\mathcal{T}(G/\{w\})$ of the subgraph starting from w corresponds to the optimal reconciliations of the DP subproblem identified by the cell (p, h, e) with which w is labeled. We define the set $\mathit{EV}(w)$ of an OR$^+$ node w to be the set of V-equivalence classes of $\mathcal{T}(G/\{w\})$ (i.e., the set $\pi(\mathcal{T}(G/\{w\}))$, where π maps a reconciliation to its event vector, see Definition 2.15). Then, the sets EV can be computed as follows (for simplicity, we will identify an OR$^+$ node with the cell (p, h, e) with which it is labeled):

- For each goal node $(p, h, \mathbb{T})$, $\mathit{EV}(p, h, \mathbb{T}) := \{(0, 0, 0, 0\}$.
- For each OR node (p, h, e), let $\{((p_1^i, h_1^i, e_1^i), (p_2^i, h_2^i, e_2^i))\}_{1\le i\le k}$ be its set of grandchild couples, then $\mathit{EV}(p, h, e)$ can be computed as

$$\bigcup_{1\le i\le k} \bigcup_{\substack{\vec{u}_1\in \mathit{EV}(p_1^i,h_1^i,e_1^i)\\ \vec{u}_2\in \mathit{EV}(p_2^i,h_2^i,e_2^i)}} \left\{ \vec{u}_1 + \vec{u}_2 + (0, 0, 0, \xi(h, h_1, h_2)) + \begin{cases} (1,0,0,0) & \text{if } e = \mathbb{C} \\ (0,1,0,0) & \text{if } e = \mathbb{D} \\ (0,0,1,0) & \text{otherwise, } e = \mathbb{S} \end{cases} \right\}. \tag{2.8}$$

The set of event vectors of $\mathcal{T}(G)$ that we seek is the union $\bigcup_w \mathit{EV}(w)$ taken over the set of start nodes of G, i.e., the OR$^+$ nodes labeled with a cell of the form $(r(P), h, e)$.

For each of the $O(nm)$ OR$^+$ nodes of the reconciliation graph, we need to keep an extra set of size $O(nm^3)$. The space complexity is therefore $O(n^2m^4)$. For each OR node and for each of its $O(n^2)$ grandchild couples, we need to compute the Cartesian sum of two sets of EV's of size $O(nm^3)$ each; this can be done naively in time $O(n^2m^6)$ (to

improve this, see, e.g., [FJ84]). The overall time complexity is $O(n^5m^7)$.

Given an event vector, finding one optimal reconciliation in that V-equivalence class can be done using a standard technique called *backtracking*. Let us define a function `Backtrack` that takes two parameters: an OR^+ node w of the reconciliation graph and a vector $\vec{v}$ satisfying $\vec{v} \in \mathit{EV}(w)$. The function outputs an optimal subproblem reconciliation $\phi_w \in \mathcal{T}(G/\{w\})$ such that $\vec{e}\,(\phi_w) = \vec{v}$. We choose to represent a reconciliation as a sequence of triples of the form (p, h, e), and the function will output one triple in each recursive step. The function `Backtrack`$(w, \vec{v})$ can be implemented as follows:

1. Let (p, h, e) be the cell with which w is labeled. Output the triple (p, h, e). If w is a goal node, stop. Otherwise, go to step 2.

2. Let $\{((p_1^i, h_1^i, e_1^i), (p_2^i, h_2^i, e_2^i))\}_{1 \le i \le k}$ be the grandchild couples of w. Find any index i such that there exist $\vec{u}_1 \in \mathit{EV}(p_1^i, h_1^i, e_1^i)$ and $\vec{u}_2 \in \mathit{EV}(p_2^i, h_2^i, e_2^i)$ such that the sum inside the big braces of Equation (2.8) is equal to $\vec{v}$ (such an i necessarily exists). Choose any such $\vec{u}_1$ and $\vec{u}_2$. Then, call `Backtrack`$((p_1^i, h_1^i, e_1^i), \vec{u}_1)$ and `Backtrack`$((p_2^i, h_2^i, e_2^i), \vec{u}_2)$.

Given a start node w and an event vector $\vec{v} \in \mathit{EV}(w)$, it suffices to call `Backtrack`$(w, \vec{v})$ to get one representative of the V-equivalence class $\vec{v}$. Finally, if we replace "any" by "all" in step 2. of `Backtrack`, we can easily adapt the algorithm so that it enumerates all reconciliations, or counts the number of reconciliations in a V-equivalence class.

2.3.2 V^*-equivalence relation

It is easy to see that the method described in the previous section for dealing with the V-equivalence relation can be applied directly to the V^*-equivalence relation. For finding all starred event vectors, the extra spaced needed for storing those is only $O(nm^3)$, as the total number of starred event vectors is bounded by $O(m^2)$; the time complexity is $O(n^3m^5)$.

2.4 Dealing with the E-, EL-, and CD-equivalence relations

2.4.1 E-equivalence relation

By Definition 2.15, an E-equivalence class is a function that maps each parasite node p to its color of the form (p, e), where e is an event type $e \in \mathcal{E} := \{\mathbb{T}, \mathbb{C}, \mathbb{D}, \mathbb{S}\}$. In this section, we will represent an E-equivalence class as a set T of colors of the form (p, e). In the same manner, a reconciliation $\phi \in \mathcal{T}(G)$ will be written as a set of triples of the form (p, h, e). Using this notation, a reconciliation ϕ is in an E-equivalence class T (i.e., $\pi(\phi) = T$) if and only if, for each triple $(p, h, e) \in \phi$, there exists a unique couple $(p, e) \in T$. A set T of couples of the form (p, e) is an E-equivalence class of the instance (i.e., $T \in \pi(\mathcal{T}(G))$) if and only if there exists $\phi \in \mathcal{T}(G)$ such that $\pi(\phi) = T$.

The problem of studying a particular E-equivalence class is easy. Given an E-equivalence class T, the reconciliation graph G can be pruned in such a way that its set of solution subtrees corresponds precisely to the reconciliations in the class T (we simply need to remove all OR nodes unless its label (p, h, e) corroborates the given class, i.e., $(p, e) \in T$). Counting and enumerating the reconciliations in that particular E-equivalence class can then be done using exactly the same method as for the "full" reconciliation graph.

Counting and enumerating the E-equivalence classes are, however, more challenging problems. At present, we will first concentrate on the enumeration of the set $\pi(\mathcal{T}(G))$ of E-equivalence classes.

Enumerating the E-equivalence classes

The algorithm is based on the simple idea of traversing the reconciliation graph in a top-down fashion (a similar approach can be used in an algorithm that enumerates all the solution subtrees). In order to obtain a polynomial time delay algorithm, during the traversal, we can no longer consider the nodes one by one; the sets of nodes that are contained in E-equivalent reconciliations must be traversed *together*. To make this clear, we will define conveniently the *color* of an OR^+ node in the reconciliation graph; the algorithm will then traverse nodes of the same color together.

Definition 2.16 (Color of an OR^+ node. Color couple)**.**

- Let w be an OR^+ node labeled by a cell $(p, h, e) \in U$, we say that w is *colored by* the

ordered pair $\kappa(w) := (p, e) \in V(P) \times \mathcal{E}$.

- Let w_1 and w_2 be two OR$^+$ nodes colored respectively by (p_1, e_1) and by (p_2, e_2). The *color couple* of the couple of nodes (w_1, w_2) is the couple of colors $((p_1, e_1), (p_2, e_2))$.

To enumerate the E-equivalence classes by a top-down recursive traversal of the reconciliation graph, the algorithm should achieve the following goal: given a set $\mathcal{O}$ of OR$^+$ nodes of the same color (p, e), enumerate $\pi(\mathcal{T}(G/\mathcal{O}))$, i.e., all E-equivalence classes of the subgraph $G/\mathcal{O}$. Any such class will include the color (p, e). If p is not a leaf, the possible events of the two children of the node p are dictated by the color couples of the grandchild couples ***gch***$(\mathcal{O})$ (by extension, ***gch*** of a set of nodes is the union of ***gch*** of every node in the set). A naive algorithm can be described as follows: for each color couple $((p_1, e_1), (p_2, e_2))$ of ***gch***$(\mathcal{O})$, first take the union $\mathcal{O}_1$ of the first grandchildren of color (p_1, e_1) and the union $\mathcal{O}_2$ of the second grandchildren of color (p_2, e_2), then call the algorithm on $\mathcal{O}_1$ and independently on $\mathcal{O}_2$, and finally combine the results together, that is, perform a Cartesian product between $\pi(\mathcal{T}(G/\mathcal{O}_1))$ and $\pi(\mathcal{T}(G/\mathcal{O}_2))$.

The pitfall of the naive approach is that not every combination between the E-equivalence classes of the reconciliations of the two child subtrees is valid. Our algorithm, shown in Algorithm 1, can be viewed as an improved version of the naive algorithm in which particular care has been taken to ensure that only valid combinations are outputted. Along with each E-equivalence class T, it also outputs a set $\widetilde{\mathcal{O}}$ which is a subset of the input set $\mathcal{O}$: it is equal the union of the root OR$^+$ nodes of all solution subtrees $\phi \in \mathcal{T}(G/\mathcal{O})$ such that $\pi(\phi) = T$. Notice that in Algorithm 1 we employ both the **return** and the **yield** statements for the output, the difference being that the latter does not halt the algorithm.

Before the proof of correctness, let us recall that we use interchangeably the terms *optimal reconciliations* and *solution subtrees*. We denote by $r(\phi)$ the root node of the solution subtree ϕ. Observe that, if ϕ is a reconciliation of (H, P, σ), then $r(\phi)$ is a start node; if ϕ is an optimal solution of the DP-subproblem indexed by a cell (p, h, e) (that is, a reconciliation of $P|_p$), then $r(\phi)$ is an OR$^+$ node colored by (p, e).

Lemma 2.2. *The function* `Enumerate`$(p, e, \mathcal{O})$ *of Algorithm 1 outputs all E-equivalence classes*

Algorithm 1: Enumerating the set of E-equivalence classes

Input: *A node* p *of the parasite tree, an event* $e \in \mathcal{E}$*, a set* $\mathcal{O}$ *of OR*$^+$ *nodes*
Require: *The nodes in* $\mathcal{O}$ *are all colored with* (p, e).
Output: *All E-equivalence classes of* $\mathrm{G}/\mathcal{O}$*, and for each class, a subset of* $\mathcal{O}$
Function `Enumerate`$(\mathrm{p}, e, \mathcal{O})$**:**
  if p *is a leaf* **then** // necessarily $e = \mathbb{T}$ and $\mathcal{O}$ only contains goal nodes
    return $\{(\mathrm{p}, e)\}$, $\mathcal{O}$
  end
  /* otherwise, necessarily $e \in \{\mathbb{C}, \mathbb{D}, \mathbb{S}\}$ and $\mathcal{O}$ only contains OR nodes */
  Let $(\mathrm{p}_1, \mathrm{p}_2)$ be the children of p
  Partition the set of grandchild couples $\boldsymbol{gch}(\mathcal{O}) := \bigcup_{w \in \mathcal{O}} \boldsymbol{gch}(w)$ according to their color couples
  for *each subset* $\{(w_1^i, w_2^i)\}_{1 \le i \le k}$ *of* $\boldsymbol{gch}(\mathcal{O})$ *of color couple* $((\mathrm{p}_1, e_1), (\mathrm{p}_2, e_2))$ **do**
    Let $\mathcal{O}_1 := \bigcup_{1 \le i \le k} \{s_1^i\}$ // $\mathcal{O}_1$ is the set of the first grandchildren
    for *each pair of* T_1 *and* $\widetilde{\mathcal{O}_1}$ *outputted by* `Enumerate`$(\mathrm{p}_1, e_1, \mathcal{O}_1)$ **do**
      Let $\mathcal{O}_2 := \bigcup_{1 \le i \le k} \left\{ w_2^i \mid \exists\, w_1 \in \widetilde{\mathcal{O}_1} \text{ s.t. } (w_1, w_2^i) \in \boldsymbol{gch}(\mathcal{O}) \right\}$
      /* $\mathcal{O}_2$ is the set of the second grandchildren compatible with $\widetilde{\mathcal{O}_1}$ */
      for *each pair of* T_2 *and* $\widetilde{\mathcal{O}_2}$ *outputted by* `Enumerate`$(\mathrm{p}_2, e_2, \mathcal{O}_2)$ **do**
        Let $\widetilde{\mathcal{O}} := \left\{ w \in \mathcal{O} \mid \exists\, (w_1, w_2) \in \widetilde{\mathcal{O}_1} \times \widetilde{\mathcal{O}_2}, \text{ s.t. } (w_1, w_2) \in \boldsymbol{gch}(w) \right\}$
        yield $\mathsf{T}_1 \cup \mathsf{T}_2 \cup \{(\mathrm{p}, e)\}$, $\widetilde{\mathcal{O}}$
      end
    end
  end

in $\pi(\mathcal{T}(\mathrm{G}/\mathcal{O}))$ *exactly once, and for each outputted pair of* T *and* $\widetilde{\mathcal{O}}$*, we have*

$$\widetilde{\mathcal{O}} = \bigcup_{\phi} \{\mathrm{r}(\phi) \mid \pi(\phi) = \mathsf{T},\ \phi \in \mathcal{T}(\mathrm{G}/\mathcal{O})\}. \tag{2.9}$$

Proof. The proof is by induction on the height h_{p} of the subtree $\mathrm{P}|_{\mathrm{p}}$. We use the fact that the pre-condition in the **Require** statement in Algorithm 1 is true for all recursive calls of `Enumerate` (easy induction). When $h_{\mathrm{p}} = 0$, p is a leaf and $\{(\mathrm{p}, \sigma(\mathrm{p}), \mathbb{T})\}$ is the

only reconciliation in $\mathcal{T}(G/\mathcal{O})$, therefore, $\{(p, e)\}$ is the only E-equivalence class. The outputted set $\mathcal{O}$ contains in this case the unique goal node of G labeled by $(p, \sigma(p), \mathbb{T})$. Now we assume $h_p > 0$.

(First direction) Consider a fixed pair of $T := T_1 \cup T_2 \cup \{(p, e)\}$ and $\widetilde{\mathcal{O}}$ outputted at Line 16, and take a node w in $\widetilde{\mathcal{O}}$. We show that there exists a reconciliation $\phi \in \mathcal{T}(G/\mathcal{O})$ such that $w = r(\phi)$ and $\pi(\phi) = T$ (in other words, T is a valid E-equivalence class in $\pi(\mathcal{T}(G))$). By the induction hypotheses, T_1 is an E-equivalence class so there exists a reconciliation ϕ_1 of $P|_{p_1}$ such that $\pi(\phi_1) = T_1$. Let $w_1 := r(\phi_1)$. Take a node $w_2 \in \mathcal{O}_2$ such that $(w_1, w_2) \in \mathit{gch}(w)$. By the induction hypotheses, there exists a reconciliation ϕ_2 of $P|_{p_2}$ such that $r(\phi_2) = w_2$ and $\pi(\phi_2) = T_2$. Define $\phi := \phi_1 \cup \phi_2 \cup \{(p, h, e)\}$, where (p, h, e) is the label of w. Then ϕ is a valid reconciliation in $\mathcal{T}(G/\mathcal{O})$ (notice that ϕ is a solution subtree of $G/\mathcal{O}$ if and only if $(w_1, w_2) \in \mathit{gch}(w)$), and satisfies $\pi(\phi) = T$.

(Second direction) Consider an E-equivalence class $T \in \pi(\mathcal{T}(G/\mathcal{O}))$, and take a reconciliation $\phi \in \mathcal{T}(G/\mathcal{O})$ such that $\pi(\phi) = T$. We show that T is outputted exactly once at Line 16 together with a set $\widetilde{\mathcal{O}}$ containing the root node of ϕ. Assume that the root node $w := r(\phi)$ is labeled with the triple (p, h, e), then ϕ can be uniquely written as the union $\phi_1 \cup \phi_2 \cup \{(p, h, e)\}$ where ϕ_1 and ϕ_2 are respectively reconciliations of $P|_{p_1}$ and $P|_{p_2}$. Furthermore, T can be uniquely written as the union $T_1 \cup T_2 \cup \{(p, e)\}$ where $T_1 = \pi(\phi_1)$ and $T_2 = \pi(\phi_2)$. Notice that T_1 and T_2 do not depend on the choice of ϕ; for T to be outputted exactly once, it suffices to show that each of T_1 and T_2 is outputted exactly once. For $i = 1, 2$, let $w_i := r(\phi_i)$ and let $(p_i, e_i) := \kappa(w_i)$ be the color of w_i. At Line 10, we only need to consider the iteration corresponding to the color couple $((p_1, e_1), (p_2, e_2))$, as no other iteration can output T_1 or T_2 from a recursive call. Since $w_1 \in \mathcal{O}_1$ and $\phi_1 \in \mathcal{T}(G/\mathcal{O}_1)$, by the induction hypotheses, T_1 is outputted exactly once in Line 12 together with a set $\widetilde{\mathcal{O}_1}$ containing w_1. For this pair of T_1 and $\widetilde{\mathcal{O}_1}$, the set $\mathcal{O}_2$ computed at Line 13 contains the node w_2. Hence, by applying again the induction hypotheses to $\phi_2 \in \mathcal{T}(G/\mathcal{O}_2)$, T_2 is outputted exactly once in Line 14 together with $\widetilde{\mathcal{O}_2}$ containing w_2. It remains to check that the set $\mathcal{O}$ outputted together with T does contain the node w. As we have $(w_1, w_2) \in \widetilde{\mathcal{O}_1} \times \widetilde{\mathcal{O}_2}$, this is straightforward from the computation of $\mathcal{O}$ (Line 15). □

Theorem 2.1. *Using Algorithm 1, the set of E-equivalence classes of a reconciliation graph*

can be enumerated in $O(mn^2)$ *time delay, where* $m = |V(P)|$ *and* $n = |V(H)|$.

Proof. To obtain all E-equivalence classes in $\pi(\mathcal{T}(G))$, it suffices to first partition the set of start nodes of the reconciliation graph according to their colors, then, for each subset $\mathcal{O}_i$ of start nodes of color (p, e), make one call of `Enumerate`$(p, e, \mathcal{O})$. By Lemma 2.2, we output every E-equivalence class of $\mathcal{T}(G/\mathcal{O})$ exactly once. Since any E-equivalence class of $\mathcal{T}(G)$ is an E-equivalence class of $\mathcal{T}(G/\mathcal{O}_k)$ for a unique k, we output every E-equivalence class of $\mathcal{T}(G)$ exactly once.

For the complexity, consider the recursion tree formed by the recursive calls of `Enumerate`. Notice that each node p of the parasite tree corresponds to exactly one recursive call, the size of the recursion tree is thus $O(m)$. In each recursive call, the partitioning of $\mathit{gch}(\mathcal{O})$ and the computation of the sets $\mathcal{O}_1$, $\mathcal{O}_2$, and $\widetilde{\mathcal{O}}$ can all be done in time linear in the size of $\mathit{gch}(\mathcal{O})$, which is $O(n^2)$. Therefore, $O(mn^2)$ time is needed in the worst case between outputting two E-equivalence classes. □

Counting the E-equivalence classes

The next algorithm is a heuristic: it computes the exact value of $|\pi(\mathcal{T}(G))|$ without a guaranteed upper bound on the running time or on the space. The idea is to build an ad-AND/OR graph with a structure similar to that of the reconciliation graph and whose solution subtrees correspond bijectively to the E-equivalence classes; after this step, counting and enumerating the E-equivalence classes can be done using the standard techniques for ad-AND/OR graphs. Just as the reconciliation graph, such a graph can have a relatively small size even when the number of solution subtrees is huge (say, 10^6 arcs in the graph versus 10^{40} solution subtress). In Chapter 3, Section 3.1.2, we will see that this heuristic method does allow us to compute some huge numbers of E-equivalence classes.

Exactly like the reconciliation graph, the graph is constructed in a bottom-up fashion using the dynamic programming technique. The DP-table is denoted by D. Each step is indexed by a cell $(p, h, e) \in U$. The space of cells is traversed following the post-order of the parasite tree. Notice that the input of the algorithm will be the reconciliation graph. For simplicity, we use the notation $M(p, h, e)$ to denote the unique OR^+

node in the reconciliation graph labeled by (p, h, e) (see Section 2.1.3, and also Equations (2.4) and (2.5)). At the end of the step (p, h, e), the value $D(p, h, e)$ of the DP-table will also be an OR$^+$ node: instead of being labeled by a triple (p, h, e), this node will be labeled by a couple (p, e) (i.e., a color, see Definition 2.16).

We will work on OR$^+$ nodes while keeping in mind the following: an OR$^+$ node w induces a complete ad-AND/OR graph structure containing all nodes that are reachable from w and in which w is the unique start node. We denote by $\mathcal{T}(w)$ the set of solution subtrees of the ad-AND/OR graph induced by the OR$^+$ node w. This "new" notation should be intuitive: if w is an OR$^+$ node in a given graph G, the set $\mathcal{T}(w)$ was denoted by $\mathcal{T}(G/\{w\})$. When we consider an OR$^+$ nodes for which the set of solution subtrees corresponds to a set of reconciliations (that is, an OR$^+$ node of the reconciliation graph), we denote by $\pi(\mathcal{T}(w))$ the set of E-equivalence classes (see also Equation (2.7), Definition 2.15):

$$\pi(\mathcal{T}(w)) := \bigcup_{\phi \in \mathcal{T}(w)} \pi(\phi). \tag{2.10}$$

Either for a reconciliation or for an E-equivalence classes, we will use its correspondence with the *label set* (or a *set of colors*) of a solution subtree: by looking at the labels of the OR$^+$ nodes in each solution subtree in $\mathcal{T}(w)$, we obtain a set of labels of the form (p, h, e) or (p, e). As we mentioned in the beginning of this section, a set of colors T is a valid E-equivalence class for the subproblem (p, h, e), i.e., $T \in \pi(\mathcal{T}(M(p, h, e)))$, if and only if there exists $\phi \in \mathcal{T}(M(p, h, e))$ such that $\pi(\phi) = T$.

Using this notation, we know that the OR$^+$ node $M(p, h, e)$ of the reconciliation graph satisfies the following:

- $\mathcal{T}(M(p, h, e))$ corresponds bijectively to the set of optimal reconciliations of the DP-subproblem indexed by (p, h, e) (those are reconciliations ϕ of the subtree $P|_p$ satisfying $\phi(p) = h$ and $E_\phi(p) = e$).

To obtain an ad-AND/OR graph as desired, the OR$^+$ node $D(p, h, e)$ of the DP-table should satisfy the following lemma, which states in particular that two different solution subtrees should not correspond to the same label set (we will show this once we will have described how to compute $D(p, h, e)$):

Lemma 2.3. $\mathcal{T}(D(p, h, e))$ *corresponds bijectively to the set of E-equivalence classes of the DP-subproblem indexed by* (p, h, e)*, that is, (1) we have the equivalence between the two sets of label sets:*

$$\mathcal{T}(D(p, h, e)) = \pi(\mathcal{T}(M(p, h, e))), \tag{2.11}$$

and (2) $\mathcal{T}(D(p, h, e))$*, as a multiset of label sets, has no redundancies.*

The DP-algorithm for computing $D(p, h, e)$ relies on a certain `Merge` operation that we will not present immediately. Without describing *how* it does it, we first state *what* it does:

Lemma 2.4. *The* `Merge` *function takes as input a list* $w_1, \ldots, w_k$ *of OR nodes satisfying:*

- (Pre-condition 1) *Every* w_i *has the same label,*
- (Pre-condition 2) *For every grandchild couple* $(w_1^i, w_2^i) \in \mathit{gch}(w_i)$*, both* $\mathcal{T}(w_1^i)$ *and* $\mathcal{T}(w_2^i)$*, as multisets of label sets, have no redundancies.*

It returns a single OR node w_m *such that*

- (Post-condition 1) *We have the equality between the two sets of label sets*

$$\mathcal{T}(w_m) = \bigcup_{i=1}^{k} \mathcal{T}(w_i)\,, \tag{2.12}$$

- (Post-conditions 2) $\mathcal{T}(w_m)$*, as a multiset of label sets, has no redundancies.*

Now, we are ready to describe how to fill up the DP-table D. Since the space of cells is the same, and the cells are filled in the same order, the algorithm is very similar to the construction of the reconciliation graph (see Section 2.1.3). For each cell $(p, h, e) \in U$ such that $M(p, h, e)$ exists, in the post-order of $V(P)$:

- If p is a leaf, $D(p, h, e)$ is a goal node labeled by (p, e).
- Otherwise, p is an internal node with children (p_1, p_2).
  - For each child AND node of $M(p, h, e)$, let (p_1, h_1, e_1) and (p_2, h_2, e_2) be the labels of its two child OR$^+$ nodes. Create an OR node w_i labeled by (p, e) with one single child AND node. Connect this AND node to the two OR$^+$ nodes $D(p_1, h_1, e_2)$ and $D(p_2, h_2, e_2)$.

– $D(p, h, e)$ is an OR node labeled by (p, e) and is equal to $\texttt{Merge}(w_1, \ldots, w_k)$, where the w_i's are the OR nodes created in the previous step.

Proof. (of Lemma 2.3) The second part, i.e., $\mathcal{T}(D(p, h, e))$ has no redundancies, is a direct result of Lemma 2.4, post-condition 2 (it can be checked by easy induction that the pre-conditions of `Merge` are satisfied). We will prove the first part of Lemma 2.3, i.e., Equation (2.11), by induction on the height h_p of the subtree $P|_p$. When $h_p = 0$, p is a leaf and $\{(p, h, e)\}$ is the only reconciliation in $\mathcal{T}(M(p, h, e))$, thus, $\{(p, e)\}$ is the only E-equivalence class. Now we assume $h_p > 0$.

(First direction) Take $T \in \mathcal{T}(D(p, h, e))$. It can be written as $T_1 \cup T_2 \cup \{(p, e)\}$ where $T_i \in \mathcal{T}(D(p_i, h_i, e_i))$, for $i = 1, 2$, for previously filled cells (p_i, h_i, e_i). By the induction hypotheses, T_1 and T_2 are valid E-equivalence classes of reconciliations of the subtrees $P|_{p_1}$ and $P|_{p_2}$. Now, the set of colors T is a valid E-equivalence class if and only if there exists an OR^+ node w colored by (p, e) in the reconciliation graph that has a grandchild couple $(w_1, w_2) \in \mathit{gch}(w)$ of color couple $((p_1, e_1), (p_2, e_2))$ (see Definition 2.16). This is true; it suffices to choose $w := M(p, h, e)$.

(Second direction) Take $\phi \in \mathcal{T}(M(p, h, e))$. It can be written as $\phi_1 \cup \phi_2 \cup \{(p, h, e)\}$ where $\phi_i \in \mathcal{T}(M(p_i, h_i, e_i))$, for $i = 1, 2$, for previously filled cells (p_i, h_i, e_i). By the induction hypotheses, there exist $T_1 \in \mathcal{T}(D(p_1, h_1, e_1))$ and $T_2 \in \mathcal{T}(D(p_2, h_2, e_2))$ such that $\pi(\phi_i) = T_i$, for $i = 1, 2$. Let $T := T_1 \cup T_2 \cup \{(p, e)\}$. It is clear that $\pi(\phi) = T$. For T to be a solution subtree, i.e., for T to be in $\mathcal{T}(D(p, h, e))$, it suffices to find an OR node w_i created during the construction step (before applying `Merge`, see also Lemma 2.4) such that $T \in \mathcal{T}(w_i)$. Such a w_i can be found, since there exists a child AND node of $M(p, h, e)$ having $(M(p_1, h_1, e_1), M(p_2, h_2, e_2))$ as children (it is the child AND node of $M(p, h, e)$ in the solution subtree ϕ). □

It remains to describe the `Merge` function. The presentation is complicated, and is deferred to the end of this chapter in Section 2.5 as Supplementary material.

2.4.2 CD-equivalence relation

To apply the methods for enumerating and counting the E-equivalence classes to the CD-equivalence relation, we only need to adapt the Definition 2.16 of the color of an OR^+ node so that it correctly reflects the color of a parasite node under a reconciliation for the CD-equivalence relation (see Definitions 2.14 and 2.15). Concretely, if w is an OR^+ node labeled by (p, h, e), its color is either (p, h, e), if $e \neq \mathbb{S}$, or $(p, ?, \mathbb{S})$ otherwise.

2.4.3 EL-equivalence relation

To apply the same methods as for the E-equivalence relation, we can build a slightly modified version of the reconciliation graph which keeps tracks of the host-switch arcs. Consider a new set of event types $\mathcal{E}' := \{\mathbb{T}, \mathbb{C}, \mathbb{D}, \mathbb{S}_L, \mathbb{S}_R\}$ (see Definition 2.14). Given a reconciliation graph G, we will modify G so that each OR^+ node, instead of being labeled by $(p, h, e) \in V(P) \times V(H) \times \mathcal{E}$, will now be labeled by $(p, h, e') \in V(P) \times V(H) \times \mathcal{E}'$. To do this, for every OR node w in G labeled by a cell (p, h, e) such that $e = \mathbb{S}$, in the post-order of $V(P)$, we perform the following operations:

1. Partition the set of grandchild couples $gch(w)$ into two (potentially empty) subsets denoted by gch_L and gch_R: for each $(w_1, w_2) \in gch(w)$, respectively labeled by (p_1, h_1, e_1) and (p_2, h_2, e_2), we put (w_1, w_2) to gch_L if $h_1 \nsucc h$, otherwise (we have necessarily $h_2 \nsucc h$), we put (w_1, w_2) to gch_R.

2. If $gch_L = \emptyset$, simply change the label of w to $(p, h, \mathbb{S}_L)$. Otherwise, if $gch_R = \emptyset$, simply change the label of w to $(p, h, \mathbb{S}_R)$. In the remaining case, that is, both gch_L and gch_R are nonempty, go to step 3.

3. Duplicate w into two nodes w_L and w_R, respectively labeled by $(p, h, \mathbb{S}_L)$ and $(p, h, \mathbb{S}_R)$ (to duplicate a node means to copy all the incoming and outgoing arcs). Remove the arcs between w_L and some of its children until $gch(w_L) = gch_L$. Remove the arcs between w_R and some of its children until $gch(w_R) = gch_R$.

After this step (which takes linear time in the size of the graph), any method for the E-equivalence relation can be used directly without modification.

2.5 Supplementary material: the `Merge` function

The goal of this section is to present the `Merge` function mentioned in Lemma 2.4. In this section, we do not work on the reconciliation graph but on some general ad-AND/OR graph represented by its start node w, and such that each of its OR$^+$ node is labeled by a color of the form (p, e). We will be interested in questions such as "*does $\mathcal{T}(w)$, as a multiset of label sets, have redundancies?*" In order to clarify and simplify this statement, we start by introduce some new notations.

2.5.1 Notations and definitions

If $T \in \mathcal{T}(w)$ is a solution subtree, we denote by $C(T)$ the set of colors of the OR$^+$ nodes in T (i.e., the label set). What we have been calling the *multiset of label sets* is denoted as the multiset $\mathcal{C}(w) := \{C(T) \mid T \in \mathcal{T}(w)\}$. Instead of talking about label sets or sets of colors, we will consider *colorings*, which we define next. Indeed, for any w that we will encounter in the algorithm, a label set in $\mathcal{C}(w)$ always coincides with a coloring of a parasite subtree.

We say that an OR$^+$ node w *colors* a node $p \in V(P)$ if its color $\kappa(w) = (p, e)$ for some $e \in \mathcal{E}$. We will also say that an OR$^+$ node *colors* a parasite subtree, using the following recursive definition:

Definition 2.17. Let $p \in V(P)$. Let w be an OR$^+$ node that colors p. We say that w *colors* the subtree $P|_p$ if it satisfies the following:

- If p is a leaf, then w is a goal node.
- If p is an internal node with children (p_1, p_2), then w is an OR node, and for each grandchild couples $(w_1, w_2) \in \mathit{gch}(w)$, w_1 colors $P|_{p_1}$ and w_2 colors $P|_{p_2}$.

Given a tree and a fixed set of colors, a *coloring* (or *node-coloring*) of the tree is a function that maps each node of the tree to a color. If w colors $P|_p$, then each solution subtree $T \in \mathcal{T}(w)$ corresponds to a coloring of the subtree $P|_p$; this coloring, which can be written as a label set, is also denoted by $C(T)$. Notice that each coloring in the multiset $\mathcal{C}(w)$ has the same *root color*, i.e., the color assigned to the root p of the subtree $P|_p$

(the root color is simply $\kappa(w)$). The next important definition replaces the statement "*the multiset* $\mathcal{C}(w)$ *has no redundancies*" by a single adjective:

Definition 2.18. Let $p \in V(P)$ and let w be an OR$^+$ node that colors $P|_p$. We say that the node w is *neat* if no two colorings in the multiset $\mathcal{C}(w)$ are the same, in other words, for any two solution subtrees T_1 and $T_2 \in \mathcal{T}(w)$,

$$C(T_1) = C(T_2) \iff T_1 = T_2.$$

The following result is easy: if an OR node w is neat, and if $(w_1, w_2) \in \mathit{gch}(w)$, then both OR$^+$ nodes w_1 and w_2 are neat.

Using the new notations, Lemma 2.4 can be re-written as follows:

Lemma 2.5 (Reformulation of Lemma 2.4). *The* `Merge` *function takes as input a list* $w_1, \ldots, w_k$ *of OR*$^+$ *nodes satisfying:*

- (Pre-condition 1) *Every* w_i *colors the same subtree* $P|_p$ *with the same root color, for some node* $p \in V(P)$*,*
- (Pre-condition 2) *Every* w_i *is neat.*

It returns a single OR$^+$ *node* w_m *such that*

- (Post-condition 1) *We have the equality between the two sets of colorings*

$$\mathcal{C}(w_m) = \bigcup_{i=1}^{k} \mathcal{C}(w_i). \tag{2.13}$$

- (Post-condition 2) w_m *is neat.*

Notice that the union operation in Equation (2.13) is the union for sets; by the pre-condition 2, every $\mathcal{C}(w_i)$ is in fact a set (multiset whose elements are of multiplicity one); the left hand side is also a set thanks to the post-condition 2.

2.5.2 Description of `Merge`

`Merge`

Informally speaking, the `Merge` function has two objectives: completeness (we must capture all colorings of the input) and neatness (we must not produce duplicates). While

the second objective seems nontrivial, there is an easy method for achieving the first objective, or Equation (2.13) (it is clear that any candidate output w_m should have the same color as any input w_i; we will omit the color of w_m and focus on its children or its grandchild couples; we also omit the trivial case where the function applies to goal nodes instead of OR nodes):

- *(Step 1 of* `Merge`*)* Create an OR node w_m and connect it to every one of the child AND nodes of the input w_i's, so that the set of children of w_m is the union of the children of the input w_i:

$$ch(w_m) = \bigcup_{i=1}^{k} ch(w_i).$$

This is exactly what we decided to do in `Merge`, followed by a second step:

- *(Step 2 of* `Merge`*)* Call `MakeNeat`(w_m) so that w_m becomes neat.

We still need to describe `MakeNeat`, but we are one step closer, because, instead of trying to capture all the colorings of the (multiple) input graphs, we now only need to modify one single graph while maintaining the same set of distinct colorings.

Lemma 2.6. *The* `MakeNeat` *function takes as input a single OR node w_0 satisfying:*

- (Pre-condition 1) *w_0 colors a subtree $P|_p$, for some node $p \in V(P)$.*
- (Pre-condition 2) *For every grandchild couple $(w_1, w_2) \in gch(w_0)$, both w_1 and w_2 are neat.*

It returns a single OR node w_m such that

- (Post-condition 1) *w_m is neat.*
- (Post-condition 2) *$\mathcal{C}(w_m)$ is equal to the set of distinct colorings in $\mathcal{C}(w_0)$.*

Notations

To give a characterization of the neatness of a node, we need some more notations. Notice that a grandchild couple $(w_1, w_2) \in gch(w_0)$ corresponds to an AND node in $ch(w_0)$. Since an AND node is uniquely identified by its two child OR^+ node, in this

section, we simply denote an AND node by a couple of OR$^+$ nodes, and we also write $(w_1, w_2) \in \mathit{ch}(w_0)$. The algorithm will create AND nodes that are *not* children of some already fixed OR nodes; their common properties are related to colorings of a parasite subtree.

Definition 2.19. Let $p \in V(P) \setminus L(P)$ with children (p_1, p_2). Let (w_1, w_2) be an AND node. We say that (w_1, w_2) *neatly colors* the subtree $P|_p$ if w_1 colors $P|_{p_1}$ and w_2 colors $P|_{p_2}$, and both w_1 and w_2 are neat.

If an AND node (w_1, w_2) neatly colors $P|_p$, we denote by $w_1 \times w_2$ the set

$$w_1 \times w_2 := \mathcal{C}(w_1) \times \mathcal{C}(w_2).$$

An element C of $w_1 \times w_2$ is called a *coloring of* $P|_p$ *with undetermined root color*. In a similar spirit as for the Cartesian products of colorings, we next introduce shorthands for the intersection and the difference of sets of colorings. If w_1 and w_2 are two neat OR$^+$ nodes that color $P|_p$, we denote, respectively, by $w_1 \cap w_2$ and $w_1 \setminus w_2$ the sets

$$w_1 \cap w_2 := \mathcal{C}(w_1) \cap \mathcal{C}(w_2), \quad w_1 \setminus w_2 := \mathcal{C}(w_1) \setminus \mathcal{C}(w_2).$$

`MakeNeat`

We can now give a characterization of the neatness of a node and present a strategy for the `MakeNeat` function. Let w_0 be the input of `MakeNeat`. It is an OR$^+$ that colors $P|_p$ for some $p \in V(P)$. If p is a leaf, then w_0 is necessarily neat. Otherwise, that is, if p is an internal node:

- If, for every two child AND nodes of w_0, indexed by i and j, and denoted respectively by (w_1^i, w_2^i) and $(w_1^j, w_2^j) \in \mathit{ch}(w_0)$ (by the pre-conditions, all child AND nodes neatly color $P|_p$), we have

$$i \neq j \implies \left(w_1^i \times w_2^i\right) \cap \left(w_1^j \times w_2^j\right) = \emptyset, \tag{2.14}$$

  then w_0 is neat.

To put it short, an OR node is neat if the sets of colorings (with undetermined root color) of its child AND nodes have pairwise empty intersections.

A straightforward strategy for `MakeNeat` is to repeatedly find an "intersecting" pair of child AND nodes and to make them "intersection-free":

- Function `MakeNeat`(w_0): while there is a pair of AND nodes (w_1^i, w_2^i) and (w_1^j, w_2^j) in $ch(w_0)$ such that $(w_1^i \times w_2^i) \cap \left(w_1^j \times w_2^j\right) \neq \emptyset$, replace those two nodes by `MakeIntersectionFree`$((w_1^i, w_2^i), (w_1^j, w_2^j))$.

Lemma 2.7. *The* `MakeIntersectionFree` *function takes as input two AND nodes* (w_1^i, w_2^i) *and* (w_1^j, w_2^j) *such that both input nodes neatly color* $P|_p$ *for some* $p \in V(P)$*, and*

$$\left(w_1^i \times w_2^i\right) \cap \left(w_1^j \times w_2^j\right) \neq \emptyset. \tag{2.15}$$

It returns a set $\mathcal{A}$ *of AND nodes* $\{(w_1^k, w_2^k)\}_k$*, all of which neatly color* $P|_p$*, and such that*

$$\bigcup_k w_1^k \times w_2^k = \left(w_1^i \times w_2^i\right) \cup \left(w_1^j \times w_2^j\right), \tag{2.16}$$

and the sets of colorings $w_1^k \times w_2^k$ *of nodes in* $\mathcal{A}$ *have pairwise empty intersections.*

Table 2.4 shows what we have chosen as the output of `MakeIntersectionFree` depending on the cases (some easy set-theoretical computation will suffice for checking that the output verifies Equation (2.16) and the condition of pairwise empty intersection). The cases represent the different *causes* for two Cartesian products of sets to be have nonempty intersection (see Equation (2.15)). The first two cases (a) and (b) utilizes the `Merge` function (here, `Merge` is used recursively, as it is called on OR^+ nodes that color a subtree $P|_{p'}$ of smaller height than $P|_p$, the latter is colored by the input nodes of the `Merge` call from which the current call of `MakeIntersectionFree` originates); the output in these two cases is only one AND node. In cases (c) – (f), the output consists of two AND nodes (however, one is potentially empty, see the paragraph after Lemma 2.8 below). Notice that the four cases (c) – (f) are not not mutually exclusive (all the cases are checked sequentially until the first applicable case is found). In the most general case, that is, case (g), all four set differences $w_1^i \setminus w_1^j$, $w_1^j \setminus w_1^j$, $w_2^i \setminus w_2^j$, and $w_2^j \setminus w_2^j$ are nonempty; the output consists of three AND nodes. In cases (c) – (g), one of the two input AND nodes does not undergo any modification and is outputted directly; the modification made on the other input node (in case (g), it is also split in two) is based on the computation of intersection and difference of OR^+ nodes (last column of Table 2.4), which we will present in the next section.

	Case	Output AND nodes	Notes
(a)	$\mathcal{C}(w_1^i) = \mathcal{C}(w_1^j)$	$\left(w_1^i, \texttt{Merge}(w_2^i, w_2^j)\right)$	
(b)	$\mathcal{C}(w_2^i) = \mathcal{C}(w_2^i)$	$\left(\texttt{Merge}(w_1^i, w_1^j), w_2^i\right)$	
(c)	$\mathcal{C}(w_1^i) \subseteq \mathcal{C}(w_1^j)$	$(w_1^i, w_2^{\text{diff}})$ and (w_1^j, w_2^j)	$\mathcal{C}(w_2^{\text{diff}}) = w_2^i \setminus w_2^j$
(d)	$\mathcal{C}(w_1^j) \subseteq \mathcal{C}(w_1^i)$	(w_1^i, w_2^i) and $(w_1^j, w_2^{\text{diff}})$	$\mathcal{C}(w_2^{\text{diff}}) = w_2^j \setminus w_2^i$
(e)	$\mathcal{C}(w_2^i) \subseteq \mathcal{C}(w_2^j)$	$(w_1^{\text{diff}}, w_2^i)$ and (w_1^j, w_2^j)	$\mathcal{C}(w_1^{\text{diff}}) = w_1^i \setminus w_1^j$
(f)	$\mathcal{C}(w_2^j) \subseteq \mathcal{C}(w_2^i)$	(w_1^i, w_2^i) and $(w_1^{\text{diff}}, w_2^j)$	$\mathcal{C}(w_1^{\text{diff}}) = w_1^j \setminus w_1^i$
(g)	none of the above applies	$(w_1^{\text{diff}}, w_2^i)$ and $(w_1^{\text{inter}}, w_2^{\text{diff}})$ and (w_1^j, w_2^j)	$\mathcal{C}(w_1^{\text{diff}}) = w_1^i \setminus w_1^j$ $\mathcal{C}(w_1^{\text{inter}}) = w_1^i \cap w_1^j$ $\mathcal{C}(w_2^{\text{diff}}) = w_2^i \setminus w_2^j$

Table 2.4: A function `MakeIntersectionFree` that satisfies Lemma 2.7. It uses the `Merge` function (see Lemma 2.5) and the `InterDiff` function (see Lemma 2.8).

Lemma 2.8. *The* `InterDiff` *function takes as input two OR⁺ nodes w_1 and w_2, such that both nodes are neat, and color $\mathrm{P}|_p$ for some $p \in V(\mathrm{P})$. It outputs two nodes w^{inter} and w^{diff} such that both nodes are neat, and*

$$\mathcal{C}(w^{inter}) = w_1 \cap w_2, \quad \mathcal{C}(w^{diff}) = w_1 \setminus w_2. \tag{2.17}$$

In the case of empty intersection or difference, the output is a special symbol `Empty`*.*

In cases (c) – (f), it can happen that the "diff" node in the output is `Empty`. This corresponds to the situation where both cases (c) and (e) or both (d) and (f) apply. Intuitively, we can say that one input AND node is entirely included in the other one as sets of colorings. In this case, the output is just one single AND node, and it is equal to one of the input nodes (the "bigger" one, or the one that includes the other).

Notice that the `MakeIntersectionFree` function needs to test for the inclusion and the equality between sets of colorings. The equality is checked by double inclusion (an improved method is discussed at the end of Section 2.5.4). The inclusion can be checked by computing the intersection and the difference:

$$\mathcal{C}(w_1) \subseteq \mathcal{C}(w_2) \iff (w_1 \cap w_2 \neq \emptyset) \wedge (w_1 \setminus w_2 = \emptyset).$$

2.5.3 Intersection and difference

In this section, an AND node can either be denoted by a single letter v or by a couple (w_1, w_2) of OR$^+$ nodes that represents its children. If $v := (w_1, w_2)$ neatly colors $P|_p$, we denote its set of colorings with undetermined root color by

$$\mathcal{C}(v) = w_1 \times w_2 := \mathcal{C}(w_1) \times \mathcal{C}(w_2).$$

Preliminaries

Let w_1 and w_2 be the two input nodes of `InterDiff`. If their colors are different, it is clear that $\mathcal{C}(w_1) \cap \mathcal{C}(w_2) = \emptyset$ and $\mathcal{C}(w_1) \setminus \mathcal{C}(w_2) = \mathcal{C}(w_1)$. Suppose now that they have the same color. If they are both goal nodes, the sets of colorings $\mathcal{C}(w_1)$ and $\mathcal{C}(w_2)$ have only one element, that is, their common color. In this case, $\mathcal{C}(w_1) \cap \mathcal{C}(w_2) = \mathcal{C}(w_1)$ and $\mathcal{C}(w_1) \setminus \mathcal{C}(w_2) = \emptyset$. Suppose now that w_1 and w_2 are both OR nodes. The idea is to express the intersection and the difference using recursive calls of `InterDiff` on OR$^+$ nodes that can be found in the grandchild couples of w_1 and w_2.

This idea turns out to work well: we are able to find a practical algorithm. Before presenting the algorithm, it is convenient to introduce a subroutine, called `AND-InterDiff`. It is a function that takes two parameters, a set $\mathcal{A}$ of AND nodes and a single AND node v_j, all of them neatly color $P|_p$ for some $p \in V(P)$. It returns two sets of AND nodes $\mathcal{A}^{\text{inter}}$ and $\mathcal{A}^{\text{diff}}$. All the sets of AND nodes (one in the input and two in the output) satisfy additionally the property of *having pairwise empty intersections* as sets of colorings (see Equation (2.14)). The output should satisfy the following:

$$\begin{aligned} &\dot{\bigcup}_{v \in \mathcal{A}^{\text{inter}}} \mathcal{C}(v) \text{ should be equal to } \left(\dot{\bigcup}_{v_i \in \mathcal{A}} \mathcal{C}(v_i)\right) \cap \mathcal{C}(v_j) = \dot{\bigcup}_{v_i \in \mathcal{A}} \left(\mathcal{C}(v_i) \cap \mathcal{C}(v_j)\right), \\ &\dot{\bigcup}_{v \in \mathcal{A}^{\text{diff}}} \mathcal{C}(v) \text{ should be equal to } \left(\dot{\bigcup}_{v_i \in \mathcal{A}} \mathcal{C}(v_i)\right) \setminus \mathcal{C}(v_j) = \dot{\bigcup}_{v_i \in \mathcal{A}} \left(\mathcal{C}(v_i) \setminus \mathcal{C}(v_j)\right). \end{aligned} \tag{2.18}$$

All the union operations in the above equations are decorated with a small dot, which indicates that those are *unions on pairwise disjoint sets*. To go from the left hand side to the right hand side of the equal signs in Equation (2.18), we have used the *right distributive* property of the intersection and the difference over the union.

`AND-InterDiff`

The function `AND-InterDiff`$(\mathcal{A}, v_j)$ can be implemented as follows:

- If $\mathcal{A} = \emptyset$, return two empty sets.

- If $|\mathcal{A}| > 1$, for each $v_i \in \mathcal{A}$, call `AND-InterDiff` on the singleton set $\{v_i\}$ and v_j. Return the union of all the outputted $\mathcal{A}^{\text{inter}}$, and the union of all the outputted $\mathcal{A}^{\text{diff}}$ (we are using the right distributive property, see Equation (2.18)). Since all the union operations are on pairwise disjoint sets that contain pairwise disjoint AND nodes, a union can simply be implemented as putting AND nodes to a list.

- In the remaining case, let v_i be the only element of $\mathcal{A}$. We have two AND nodes $v_i := (w_1^i, w_2^i)$ and $v_j := (w_1^j, w_2^j)$ such that both neatly color $P|_p$ for some $p \in V(P)$, and we want to compute their intersection and difference. To do this, we will use the `InterDiff` function for two OR$^+$ nodes. Let w_1^{inter} and w_1^{diff} be the output of `InterDiff`(w_1^i, w_1^j). Let w_2^{inter} and w_2^{diff} be the output of `InterDiff`(w_2^i, w_2^j).

  - If neither w_1^{inter} nor w_2^{inter} is `Empty`, return $\mathcal{A}^{\text{inter}} := \{(w_1^{\text{inter}}, w_2^{\text{inter}})\}$. Otherwise, return $\mathcal{A}^{\text{inter}} := \emptyset$.

  - (1) If w_1^{diff} is not `Empty` and w_2^{diff} is `Empty`, return $\mathcal{A}^{\text{diff}} := \{(w_1^{\text{diff}}, w_2^i)\}$.

    (2) If w_1^{diff} is `Empty` and w_2^{diff} is not `Empty`, return $\mathcal{A}^{\text{diff}} := \{(w_1^i, w_2^{\text{diff}})\}$.

    (3) Otherwise, that is, neither w_1^{diff} nor w_2^{diff} is `Empty`,

    (3a) If either w_1^{inter} or w_2^{inter} is `Empty` (or both), return $\mathcal{A}^{\text{diff}} := \{(w_1^i, w_2^i)\}$.

    (3b) Otherwise, return $\mathcal{A}^{\text{diff}} := \left\{(w_1^{\text{diff}}, w_2^i),\ (w_1^{\text{inter}}, w_2^{\text{diff}})\right\}$.

Similar to what we did in `MakeIntersectionFree`, the different cases of `AND-InterDiff` can be understood by asking in which situations two Cartesian products have nonempty set difference; the correctness of the output can be verified by some easy set-theoretical computation. Notice that only the case (3b) requires us to check carefully that the two AND nodes in $\mathcal{A}^{\text{diff}}$ have empty intersection.

`InterDiff`

Algorithm 2 shows the function `InterDiff`. The idea that we mentioned in the beginning of this section corresponds to the **for** loop at Lines 13–21. Every occurrence of the $\cup$ operator corresponds to the union between disjoint sets of AND nodes, and the AND nodes in the same set always have pairwise empty intersection. The variable $\mathcal{A}_i$ records the difference between v_i, the i-th child AND node of w_1 and the union of the first j child AND nodes of w_2 (here, the sets of children have an arbitrary ordering), and is equal to $\mathcal{A}^{\text{diff}}_{i,j}$ during the inner loop (Line 18), for $1 \leq j \leq |ch(w_2)|$. We have, at Line 16, if $j > 1$:

$$\mathcal{C}(\mathcal{A}^{\text{diff}}_{i,j}) = \mathcal{C}(\mathcal{A}^{\text{diff}}_{i,j-1}) \setminus \mathcal{C}(v_j) = \left(\mathcal{C}(v_i) \setminus \left(\bigcup_{1\leq k\leq j-1} \mathcal{C}(v_k)\right)\right) \setminus \mathcal{C}(v_j) = \mathcal{C}(v_i) \setminus \left(\bigcup_{1\leq k\leq j} \mathcal{C}(v_k)\right).$$

If $j = 1$, $\mathcal{C}(\mathcal{A}^{\text{diff}}_{i,j}) = \mathcal{C}(v_i) \setminus \mathcal{C}(v_j)$. The intersection between the i-th child v_i of w_1 and the j-th child of w_2 can also be expressed using the intersection between v_i and the first j children of w_2 (before computing the intersection with v_j, we first subtract some previous child AND nodes that are disjoint with v_j; this does not change the intersection). We have, at Line 16, if $j > 1$:

$$\mathcal{C}(\mathcal{A}^{\text{inter}}_{i,j}) = \mathcal{C}(\mathcal{A}^{\text{diff}}_{i,j-1}) \cap \mathcal{C}(v_j) = \left(\mathcal{C}(v_i) \setminus \left(\bigcup_{1\leq k\leq j-1} \mathcal{C}(v_k)\right)\right) \cap \mathcal{C}(v_j) = \mathcal{C}(v_i) \cap \mathcal{C}(v_j).$$

If $j = 1$, $\mathcal{C}(\mathcal{A}^{\text{inter}}_{i,j}) = \mathcal{C}(v_i) \cap \mathcal{C}(v_j)$. At the end of the inner loop, the following sets are added (using unions of disjoint sets) to $\mathcal{A}^{\text{inter}}$ and $\mathcal{A}^{\text{diff}}$ that collects all the intersection and difference between v_i, the i-child of w_1, and all the children of w_2:

$$\bigcup_{j=1}^{|ch(w_2)|} \mathcal{C}(\mathcal{A}^{\text{inter}}_{i,j}) = \bigcup_{j=1}^{|ch(w_2)|} \left(\mathcal{C}(v_i) \cap \mathcal{C}(v_j)\right) = \mathcal{C}(v_i) \cap \left(\bigcup_{j=1}^{|ch(w_2)|} \mathcal{C}(v_j)\right)$$

$$\mathcal{C}\left(\mathcal{A}^{\text{diff}}_{i,|ch(w_2)|}\right) = \mathcal{C}(v_i) \setminus \left(\bigcup_{j=1}^{|ch(w_2)|} \mathcal{C}(v_j)\right).$$

The above observations, together with the right distributive property of intersection and union, show that we indeed get the correct sets of colorings for $\mathcal{A}^{\text{inter}}$ and $\mathcal{A}^{\text{diff}}$, at

the end of the outer loop (Line 21):

$$\mathcal{C}(\mathcal{A}^{\text{inter}}) = \bigcup_{i=1}^{|ch(w_1)|} \left(\bigcup_{j=1}^{|ch(w_2)|} \mathcal{C}(\mathcal{A}^{\text{inter}}_{i,j}) \right) = \left(\bigcup_{i=1}^{|ch(w_1)|} \mathcal{C}(v_i) \right) \cap \left(\bigcup_{j=1}^{|ch(w_2)|} \mathcal{C}(v_j) \right)$$

$$\mathcal{C}(\mathcal{A}^{\text{diff}}) = \bigcup_{i=1}^{|ch(w_1)|} \mathcal{C}\left(\mathcal{A}^{\text{diff}}_{i,|ch(w_2)|} \right) = \left(\bigcup_{i=1}^{|ch(w_1)|} \mathcal{C}(v_i) \right) \setminus \left(\bigcup_{j=1}^{|ch(w_2)|} \mathcal{C}(v_j) \right).$$

These two sets consist of colorings with undetermined root color. At the end of Algorithm 2, we add the correct root color and create the two OR nodes having respectively $\mathcal{A}^{\text{inter}}$ and $\mathcal{A}^{\text{diff}}$ as sets of children. We then obtain the desired result as stated in Lemma 2.8, Equation (2.17):

$$\mathcal{C}(w^{\text{inter}}) = w_1 \cap w_2, \quad \mathcal{C}(w^{\text{diff}}) = w_1 \setminus w_2.$$

2.5.4 Another operation on a set of AND nodes

We have already finished the description of all the components of the `Merge` function. The idea was quite simple: after a "naive merge" of the input AND nodes, we remove all the duplicates in the set of colorings by making every pair of AND nodes "intersection-free". This basic operation, called `MakeIntersectionFree`, can be easily implemented given that we know how to compute the intersection and the difference between two OR^+ nodes. This is done by the `InterDiff` function which builds the set of child AND nodes of the two output nodes (intersection and difference nodes) in a nested loop over the children of the input nodes.

The `InterDiff` function is therefore the most basic operation and is needed at least $O(n^2)$ times, where n is the number of child AND nodes in all the input nodes of `Merge`. The only nontrivial operation in the nested loop of `InterDiff` is the subroutine called `AND-InterDiff` which applies `InterDiff` recursively while iterating through a set of AND nodes. The performance of `InterDiff` is sensitive to the *size* of those sets of AND nodes. Any set of AND nodes that are considered in the algorithm contains only AND nodes that are pairwise disjoint as sets of colorings, and it is constructed in two ways, either (1) by initializing with a small number of AND nodes, or (2) by performing the union operation between pairwise disjoint sets. For the purpose of improving the performance and reducing the memory required for storing the result AND/OR graph, we

Algorithm 2: Intersection and difference between two OR$^+$ nodes

Input: *Two neat OR$^+$ nodes that color* $P|_p$ *for some* $p \in V(P)$.
Output: *Two neat OR$^+$ nodes* w^{inter} *and* w^{diff}.
Function `InterDiff`(w_1, w_2)**:**
  if w_1 *and* w_2 *do not have the same color, that is,* $\kappa(w_1) \neq \kappa(w_2)$ **then**
    return $w^{\text{inter}} :=$ `Empty` and $w^{\text{diff}} := w_1$
  end
  Let $p \in V(P)$ such that w_1 and w_2 color $P|_p$
  if p *is a leaf* **then**
    return $w^{\text{inter}} := w_1$ and $w^{\text{diff}} :=$ `Empty`
  end
  Let (p_1, p_2) be the children of p
  Let $\mathcal{A}^{\text{inter}}$ and $\mathcal{A}^{\text{diff}}$ be empty sets
  for *each child AND node* $v_i \in ch(w_1)$ **do**
    $\mathcal{A}_i \leftarrow \{v_i\}$
    for *each child AND node* $v_j \in ch(w_2)$ **do**
      Let $\mathcal{A}^{\text{inter}}_{i,j}$ and $\mathcal{A}^{\text{diff}}_{i,j}$ be the output of `AND-InterDiff`$(\mathcal{A}_i, v_j)$
      $\mathcal{A}^{\text{inter}} \leftarrow \mathcal{A}^{\text{inter}} \cup \mathcal{A}^{\text{inter}}_{i,j}$
      $\mathcal{A}_i \leftarrow \mathcal{A}^{\text{diff}}_{i,j}$
    end
    $\mathcal{A}^{\text{diff}} \leftarrow \mathcal{A}^{\text{diff}} \cup \mathcal{A}_i$
  end
  Let w^{inter} be an OR node with $\kappa(w^{\text{inter}}) := \kappa(w_1)$ and $ch(w^{\text{inter}}) := \mathcal{A}^{\text{inter}}$
  Let w^{diff} be an OR node with $\kappa(w^{\text{diff}}) := \kappa(w_1)$ and $ch(w^{\text{diff}}) := \mathcal{A}^{\text{diff}}$
  `/*` If $\mathcal{A}^{\text{inter}}$ or $\mathcal{A}^{\text{diff}}$ is empty, the corresponding output is set to `Empty` `*/`
  return w^{inter} and w^{diff}

now introduce another operation, called `RemovePartners`, that decreases the size of a set of AND nodes (without changing, of course, the sets of colorings). It can be used anywhere in `AND-InterDiff` and `InterDiff`, immediately after a union operation. It

can also be used in the function `MakeNeat` which also operates on a set of AND nodes: after repeatedly applying `MakeIntersectionFree` so that there are no intersecting pair of AND nodes, `RemovePartners` can be used to reduce the number of children of the output OR node.

Given a set $\mathcal{A}$ of AND nodes, the `RemovePartners` operation repeatedly finds a pair of AND nodes satisfying certain conditions, called a *partner pair*, and replaces them by a single AND node, and it stops when no partner pair can be found. By assumption, any two distinct AND nodes (w_1^i, w_2^i) and (w_1^j, w_2^j) in $\mathcal{A}$ satisfy the condition of being *disjoint as sets of colorings*, that is,

$$\left(w_1^i \times w_2^i\right) \cap \left(w_1^j \times w_2^j\right) = \emptyset. \tag{2.19}$$

Table 2.5 shows the two cases in which two AND nodes in $\mathcal{A}$ form a partner pair, and gives the new AND node that replaces them. The $\dot{\cup}$ symbol means to perform a "naive merge" between two OR nodes of the same color by taking the union of their child AND nodes (this is the same as the first step of `Merge`; the second step of `Merge` is unnecessary here, because the sets of colorings of the two OR nodes are necessarily disjoint). Notice that the first two cases of `MakeIntersectionFree` also deal with similar situations (there, however, the second step of `Merge` cannot be omitted).

Case	To be replaced by
$\mathcal{C}(w_1^i) = \mathcal{C}(w_1^j)$ and $\kappa(w_2^i) = \kappa(w_2^j)$	$(w_1^i\,,\ w_2^i \,\dot{\cup}\, w_2^j)$
$\mathcal{C}(w_2^i) = \mathcal{C}(w_2^j)$ and $\kappa(w_1^i) = \kappa(w_1^j)$	$(w_1^i \,\dot{\cup}\, w_1^j\,,\ w_2^i)$

Table 2.5: The two cases in which two AND nodes (w_1^i, w_2^i) and (w_1^j, w_2^j) satisfying Equation (2.19) form partner pair, and the corresponding replacement node in the `RemovePartners` function.

In `RemovePartners`, we need to check for the equality between the sets of colorings of two OR nodes. Unlike `MakeIntersectionFree` which checks the equality using the double inclusion criterion (by computing the intersection and the difference between several pairs of OR$^+$ nodes), `RemovePartners`, being an optional operation by nature, does not require an exact equality check with double inclusion. A sufficient condition

for $\mathcal{C}(w_1) = \mathcal{C}(w_2)$ is that the two graphs *are isomorphic with an isomorphism that preserves the colors*. This condition can be checked in linear time in the size of the graph (or faster in practice using hashing techniques), as we can see in the following recursive definition.

Definition 2.20. Given two neat OR^+ nodes w_1 and w_2 that color $P|_p$ for some $p \in V(P)$, we say that they are *sufficiently equal* if $\kappa(w_1) = \kappa(w_2)$ and, either

1. p is a leaf (so w_1 and w_2 are goal nodes), or
2. w_1 and w_2 have the same number of children, and $\mathit{ch}(w_1)$ and $\mathit{ch}(w_2)$ can be ordered in such a way that for each $i = 1, \ldots, |\mathit{ch}(w_1)|$, the i-th child (w_1^1, w_2^1) of w_1 and the i-th child (w_1^2, w_2^2) of w_2 satisfy
   - w_1^1 and w_1^2 are sufficiently equal, and
   - w_2^1 and w_2^2 are sufficiently equal.

In the implementation, `MakeIntersectionFree` can also gain an improved performance by checking the equality (for the first two cases) in two phases: first we check the sufficient equality, and only if the first test fails, we check the double inclusion.

Chapter 3

Phylogenetic tree reconciliation: practical contributions

3.1 Experimental results

We implemented in Python the algorithms for enumerating and counting the equivalence classes from Chapter 2. In this section, we apply them on biological datasets and give some experimental results that demonstrate their usefulness in practice. The experiments are run on a laptop PC with Intel i5-3380M CPU (2.90 GHz, 4 cores) and 8 GB RAM.

3.1.1 Instances

An instance of the RECONCILIATION PROBLEM consists of a dataset, that is a triple (H, P, σ), where H and P are phylogenetic trees and $\sigma : L(P) \to L(H)$ is a function, and a cost vector $\vec{c} := (c(\mathbb{C}), c(\mathbb{D}), c(\mathbb{S}), c(\mathbb{L}))$ that gives the cost value that we associate with each event type among cospeciation, duplication, host-switch, and loss.

Each dataset that we used is identified by an alphanumeric code. The code, the size (measured as the number of nodes in the trees H and P), and the source of each dataset are given in Table 3.1. All the datasets have already been used in the literature (see [BAK13; Don+15]) and together cover different situations (sizes and topologies of the

trees) and different contexts: the COG datasets are from a genes-species context, while all the other datasets are from a hosts-parasites or hosts-symbionts context.

For each dataset, we formed five instances using five different cost vectors: $(-1, 1, 1, 1)$, for maximizing the cospeciation; $(0, 1, 1, 1)$, for minimizing the events that lead to incongruencies between the tree topologies; $(0, 1, 2, 1)$ and $(0, 2, 3, 1)$, where host-switches are more penalized; $(0, 1, 1, 0)$, which is a vector chosen only for theoretical purposes and does not penalize cospeciations and losses.

With 15 datasets and 5 cost vectors, we have a total of 75 instances.

Dataset	$\|V(H)\|$	$\|V(P)\|$	**Notes and references**
EC	13	19	*Encyrtidae* (parasitic wasps) and *Coccidae* [Den+13]
GL	15	19	Gophers and Lice [HN88; HP95]
SC	21	27	Seabirds and Chewing Lice [PPG03]
RP	25	25	Rodents and Pinworms [Hug03]
SFC	29	31	Anther-smut fungi and their Caryophyllaceous hosts [Ref+08]
PML PMP	35	35	Pelican seabirds and Lice, the trees are generated using either a maximum likelihood (L) or a maximum parsimony (P) approach [Hug+07]
FD	39	101	Fish and *Dactylogyrus* [BMB13]
RH	67	83	Rodents and Hantaviruses [RHC09]
PP	71	81	Primates and Pinworm [Hug99]
COG2085 COG3715 COG4964 COG4965	199	87 79 53 59	Genes and species trees from the Clusters of Orthologous Groups of proteins (COG) database. The unrooted gene trees are rooted in such a way that the minimum cost of the RECONCILIATION PROBLEM is minimized among all choices of root. [Tat+00; DA11]
WOLB	773	773	*Wolbachia* (bacteria) and their arthropod hosts [Sim+11; Sim12]

Table 3.1: All the datasets that were used in the experiments and their references.

3.1.2 Numbers of equivalence classes

Introduction

The first set of experiments is concerned with the number of equivalence classes of the set of optimal reconciliations, for each equivalence relation (this number is denoted by $|\pi(\mathcal{T}(G))|$ in Chapter 2). As we mentioned in Chapter 2, Section 2.2.2, for an equivalence relation to be of practical interest, we typically expect the number of equivalence classes to be significantly smaller than the number of optimal reconciliations, that is,

$$|\pi(\mathcal{T}(G))| \ll |\mathcal{T}(G)|,$$

where $\ll$ denotes "much smaller than". Indeed, if the number of equivalence classes were close to the number of optimal optimal reconciliations, analyzing one representative per equivalence class would not be advantageous, as analyzing all optimal reconciliations would require a comparable level of human effort but provides more information.

Notice that, even *without* performing the enumeration of equivalence classes or the enumeration of the representatives, the number of equivalence classes itself can still be used as a *diversity measure*. Consider for example the following situation: if we have a dataset that yields two drastically different numbers of equivalence classes (one is very small, while the other one is very large), say the numbers of E-equivalence classes, for two "similar" cost vectors, say $(0, 1, 1, 1)$ and $(0, 1, 2, 1)$, this may indicate that the solution space (i.e., the set of optimal reconciliations) for this dataset is *sensitive to the choice of the cost vector*. The issue of sensitivity to event costs has been studied literature, in particular in [BAK13], where the authors have also been interested in the consistency of *event type assignments* when different cost vectors are used (those assignments correspond to the E-equivalence classes). A small number of E-equivalence classes would indicate that the consistency is high. In [BAK13], the authors tried to discover different event type assignments (i.e., different E-equivalence classes) by sampling the solution space uniformly at random. In comparison, our algorithm for enumerating the E-equivalence classes can answer the consistency question exactly.

Results

Dataset	Cost vector	#Optimal reconciliations	#V-(V^*-)equiv. classes	#E-equiv. classes	#EL-equiv. classes	#CD-equiv. classes
EC	$(-1,1,1,1)$	2	1	1	2	2
	$(0,1,1,1)$	16	5	8	16	9
	$(0,1,2,1)$	18	6	10	18	18
	$(0,2,3,1)$	16	4	8	16	16
	$(0,1,1,0)$	24	8 (5)	16	24	24
GL	$(-1,1,1,1)$	2	1	1	2	2
	$(0,1,1,1)$	2	1	1	2	2
	$(0,1,2,1)$	2	1	1	2	2
	$(0,2,3,1)$	2	1	1	2	2
	$(0,1,1,0)$	12	5 (3)	6	9	9
SC	$(-1,1,1,1)$	1	1	1	1	1
	$(0,1,1,1)$	1	1	1	1	1
	$(0,1,2,1)$	1	1	1	1	1
	$(0,2,3,1)$	1	1	1	1	1
	$(0,1,1,0)$	113	18 (4)	10	18	24
RP	$(-1,1,1,1)$	3	1	2	3	3
	$(0,1,1,1)$	18	3	7	18	10
	$(0,1,2,1)$	3	1	2	3	3
	$(0,2,3,1)$	3	1	2	3	3
	$(0,1,1,0)$	117	30 (5)	20	35	42
SFC	$(-1,1,1,1)$	40	1	3	16	3
	$(0,1,1,1)$	184	2	6	160	6
	$(0,1,2,1)$	40	1	3	16	3
	$(0,2,3,1)$	40	1	3	16	3
	$(0,1,1,0)$	6332	110 (9)	70	363	888

PML	$(-1,1,1,1)$	2	1	1	2	1
	$(0,1,1,1)$	180	4	7	160	9
	$(0,1,2,1)$	2	1	1	2	1
	$(0,2,3,1)$	11	2	2	6	4
	$(0,1,1,0)$	448	17 (6)	24	119	119
PMP	$(-1,1,1,1)$	2	1	1	2	1
	$(0,1,1,1)$	2	1	1	2	1
	$(0,1,2,1)$	2	1	1	2	1
	$(0,2,3,1)$	18	2	2	18	10
	$(0,1,1,0)$	262	34 (6)	18	98	232
FD	$(-1,1,1,1)$	944	8	14	368	18
	$(0,1,1,1)$	25184	11	52	22752	72
	$(0,1,2,1)$	408	10	20	180	20
	$(0,2,3,1)$	80	2	2	16	2
	$(0,1,1,0)$	10^{15}	2146 (20)	54336	604980	10^{13}
RH	$(-1,1,1,1)$	1052	8	12	176	64
	$(0,1,1,1)$	42	4	4	42	8
	$(0,1,2,1)$	2208	18	38	368	196
	$(0,2,3,1)$	288	4	6	48	48
	$(0,1,1,0)$	4080384	275 (13)	1152	5832	557928
PP	$(-1,1,1,1)$	144	2	2	144	72
	$(0,1,1,1)$	5120	4	8	4480	48
	$(0,1,2,1)$	72	2	2	72	36
	$(0,2,3,1)$	72	2	2	72	36
	$(0,1,1,0)$	498960	134 (12)	1152	56700	124740
COG2085	$(-1,1,1,1)$	109056	3	6	7360	6
	$(0,1,1,1)$	44544	3	4	36224	4
	$(0,1,2,1)$	37568	8	14	3200	14
	$(0,2,3,1)$	46656	4	4	1344	10
	$(0,1,1,0)$	10^{11}	930 (13)	1152	52920	32864832

COG3715	$(-1,1,1,1)$	63360	6	16	2520	32
	$(0,1,1,1)$	1172598	28	792	777030	872
	$(0,1,2,1)$	9	2	2	7	2
	$(0,2,3,1)$	33	2	2	2	2
	$(0,1,1,0)$	10^{12}	878 (15)	2496	5976	8694960
COG4964	$(-1,1,1,1)$	36	1	1	4	1
	$(0,1,1,1)$	224	2	2	224	2
	$(0,1,2,1)$	36	1	1	4	1
	$(0,2,3,1)$	54	2	2	6	2
	$(0,1,1,0)$	8586842	376 (11)	64	220	38104
COG4965	$(-1,1,1,1)$	44800	5	13	23456	13
	$(0,1,1,1)$	17408	2	4	17408	4
	$(0,1,2,1)$	640	2	3	576	3
	$(0,2,3,1)$	6528	3	5	448	5
	$(0,1,1,0)$	907176	324 (10)	12	17	11958
WOLB	$(-1,1,1,1)$	10^{47}	10	4080	*	24192
	$(0,1,1,1)$	10^{48}	11	40960	*	76800
	$(0,1,2,1)$	10^{47}	10	4080	*	24192
	$(0,2,3,1)$	10^{42}	7	96	10^{36}	1152
	$(0,1,1,0)$	10^{136}	* (74)	10^{27}	*	*

Table 3.2: The number of optimal reconciliations and the number of equivalence classes for each equivalence relation and for each instance. For numbers larger than one billion, only the order of magnitude is shown (as powers of 10). For the instances with the cost vector $(0,1,1,0)$, the numbers of V- and V^*-equivalence classes do not coincide (see Section 2.2.3); the latter is shown in parentheses. The * symbol indicates that the counting of the equivalence classes exceeded the memory limit of 8 GB.

Discussion

First, let us look at the third column of Table 3.2, corresponding to the numbers of V- and V*-equivalence classes. We have shown in Section 2.3.1 that these numbers are bounded by a polynomial of the input size. They are indeed always very small; the only exception is that, when the cost vector is $(0, 1, 1, 0)$, the number of V-equivalence classes can get large: this can be understood easily since, by setting the loss cost $c(\mathbb{L})$ to zero, we allow more variability in the number of losses of the solutions. One can argue that, if the loss cost is set to zero, it is more reasonable to consider the V*-equivalence relation rather than the V-equivalence relation.

The numbers of E- and CD-equivalence classes are also very small except when the cost vector is $(0, 1, 1, 0)$, in this case those numbers can be relatively larger. For example, for the dataset COG2085 and cost vector $(0, 1, 1, 1)$, there are 44544 optimal reconciliations but only 4 CD-equivalence classes. This result suggests that the associations that are involved in host-switches (in this case, horizontal gene transfer events) can account for almost all the diversity of the solution space.

The numbers of EL-equivalence classes are generally larger than the other equivalence classes. For some of the instances, we can observe that while the number of E-equivalence classes is small, the number of EL-equivalence classes can be large and close to (or even equal to) the number of optimal reconciliations. For example, for the dataset PP and cost vector $(-1, 1, 1, 1)$, there are only 2 E-equivalence classes whereas all the 144 optimal reconciliations are in different EL-equivalence classes. This indicates that, compared to the E-equivalence classes that tell us which nodes are involved in host-switches, the EL-equivalence classes, which distinguish which arcs of the parasite tree are involved in host-switches, can convey much more information.

To better measure and visualize how small the number of equivalence classes are relative to the number of optimal reconciliations, we computed, for 29 instances having between 10^2 and 10^8 optimal reconciliations, a value that we call Reduction: it is equal to the number of equivalence classes divided by the number of optimal reconciliations. In Figure 3.1, each x coordinate corresponds to an instance, where the instances are sorted in the increasing order of the number of optimal reconciliations (the

denominator of Reduction); for each instance, we plotted three points that correspond to the Reduction values for the E-, EL-, and CD-equivalence relations. Figure 3.1 illustrates the observations that we made in the previous paragraphs: the E- and CD-equivalence relations correspond generally to smaller Reduction values, whereas the EL-equivalence relation yields larger Reduction values (closer to 1).

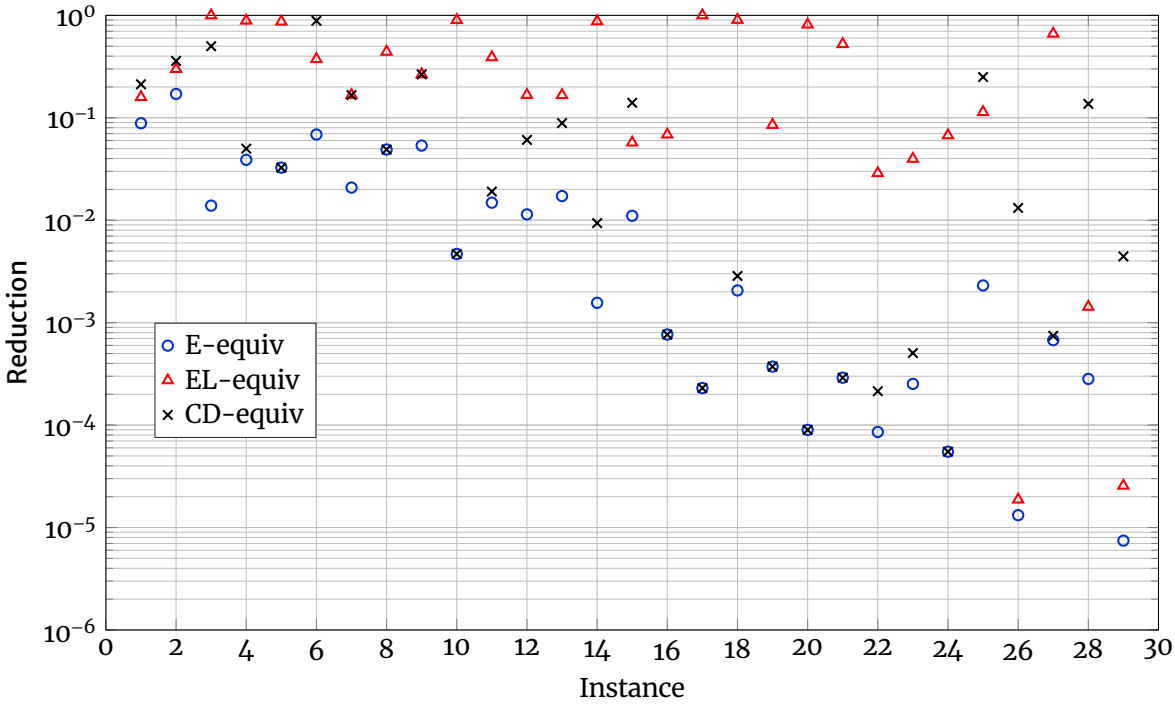


Figure 3.1: X-axis: The 29 instances in increasing order of the number of optimal reconciliations. Y-axis: In logarithmic scale, the Reduction value that is equal to the number of equivalence classes over the number of optimal reconciliations. For each instance, three points are plotted: the blue circle, the red triangle, and the black X, corresponding respectively to the number of E-, EL-, and CD-equivalence classes.

3.1.3 Lists of equivalence classes or representatives

As we have seen in the previous section, for many of the biological instances, the number of equivalence classes is small enough so that we can enumerate them all. We will now see that the list of equivalence classes can provide useful information about the solution space. For this purpose, we use the dataset WOLB that has the following particularity: all the cost vectors that we used lead to a number of optimal reconciliations

that is at least 10^{42}, which is too large for any exhaustive enumeration method. To gain insight into the solution space of this dataset, apart from the methods based on random sampling [BAK13; Don+15], only a limited number of the preexisting exact methods can be applied in practice (e.g., computing the diameter of the solution space [Haa+19], computing the distribution of the pairwise distances [SML19], computing a median reconciliation [Ngu+13], etc.). The enumeration of the equivalence classes or of the representatives of the equivalence classes is an addition to the list of exact methods, and we will see that it can be quite informative. Consider a metric m on the solution space that measures the differences in the event type assignments between reconciliations:

$$m(\phi_1, \phi_2) := \left|\{p \in V(P) \mid E_{\phi_1}(p) \neq E_{\phi_2}(p)\}\right|. \tag{3.1}$$

Then, knowing the list of E-equivalence classes and the size of each class, we can compute the diameter, the distribution of pairwise distances, as well as a median reconciliation of the solution space with respect to the metric m.

For the experiments, we choose the instance with the WOLB dataset and the cost vector $(0, 2, 3, 1)$. First, we look at the list of V-equivalence classes (i.e., the event vectors), shown in Table 3.3. The table also indicates the size of each class as proportions of the solution space (i.e., the proportion of optimal reconciliations in each V-equivalence class among all optimal reconciliations).

Event vector	**Proportion of the solution space**
$(105, 0, 281, 45)$	36.5425 %
$(106, 0, 280, 48)$	29.5704 %
$(104, 0, 282, 42)$	18.7570 %
$(107, 0, 279, 51)$	10.5588 %
$(103, 0, 283, 39)$	3.1628 %
$(108, 0, 278, 54)$	1.3807 %
$(102, 0, 284, 36)$	0.0277 %

Table 3.3: The V-equivalence classes of the instance WOLB-$(0, 2, 3, 1)$ and their size, as proportions of the solution space, sorted in the decreasing order of the size.

From the list of event vectors, one can see that the dataset can be explained by a large number of host-switches and cospeciations, and that there have probably been no duplication. The diversity of the solution space in terms of the event vectors is low, not only because the number of different event vectors is small, but also because these event vectors are very similar to each other. Furthermore, if we view the proportions of the event vectors as probability densities, the distributions of the solutions in the spaces of all four dimensions of the event vectors are *unimodal*. For example, along the axis of the number of cospeciations, solutions that have 105 (the most common value) cospeciations or a number close to 105 are the most probable; in contrast, we are less likely to find solutions that have a number of cospeciations which is far away from 105 (extreme values).

Now, we turn to the list of the 96 E-equivalence classes. Recall that an E-equivalence class is a function π that associates each parasite node $p \in V(P)$ with a color of the form (p, e), where $e \in \mathcal{E}$ is an event type. One way to measure how dissimilar two E-equivalence classes are from each other is to take the sum of differences:

$$m(\pi_1, \pi_2) := |\{p \in V(P) \mid \pi_1(p) \neq \pi_2(p)\}|. \tag{3.2}$$

Let s_i denote the size of an E-equivalence class π_i (i.e., the number of optimal reconciliations in that class). Equation (3.2) defines a measure on the set of E-equivalence classes which can be used to compute, for instance, the *average pairwise distance* of the solution space with respect to the metric in Equation (3.1) by noticing the following relationship:

$$\sum_{i \neq j} m(\phi_i, \phi_j) = \sum_{i \neq j} m(\pi_i, \pi_j) \cdot s_i \cdot s_j, \tag{3.3}$$

where the first sum is over the set of optimal reconciliations, the second sum is over the set of E-equivalence classes. We computed the distances between the E-equivalence classes. The maximum distance is 14. The average pairwise distance of the solution space (see Equation (3.3)) is around 2. Informally, it means that if we randomly select two optimal reconciliations, we expect to find two differences in their node-wise event type assignments. Those distance values are extremely small compared to the theoretical maximum of 386 (the number of internal nodes of the parasite tree). Another observation that can be made from the list of E-equivalence classes is about the

parasite nodes that can receive different event types across the solution space. Instead of looking at the total number of differences, we can ask which nodes contribute to the differences. There are only 15 of them, in other words, all other 371 internal nodes receive a consistent event type across the entire solution space. We have further confirmed that the diversity of the solution space is low: not only the event vectors are similar, the distributions of the events on the nodes of the parasite tree are also similar.

For the CD-equivalence classes, we can ask the similar question of which nodes contribute to the differences. Here, a parasite node p contributes to the differences between two CD-equivalence classes if its color (see Definition 2.14) is different under the two classes, i.e., either it receives different event types, or, if its event type is $\mathbb{C}$ or $\mathbb{D}$, it is associated with different host nodes. After enumerating and analyzing the 1152 CD-equivalence classes, we found 20 such nodes. In other words, apart from the 15 nodes that contribute to the difference in event types, there are 5 cospeciation-duplication nodes that are associated with different host nodes across the solution space. This number is again very small, which backs up the statement about the diversity of the solution space being low.

3.1.4 Finding time-feasible reconciliations using EL-equivalence classes

In Chapter 2, Section 2.1.2, we mentioned that a biologically meaningful reconciliation should satisfy the requirement of time-feasibility. Unfortunately, the TIME-FEASIBLE RECONCILIATION PROBLEM, i.e., finding a time-feasible reconciliation that minimizes the cost, is NP-hard [Ova+11; THL11]. A practical way of obtaining time-feasible reconciliations is to first enumerate the optimal solution of RECONCILIATION PROBLEM (where the time-feasible requirement is entirely dropped), and then filter out those that are not time-feasible. If we only want to obtain one time-feasible optimal reconciliation or check if there is any, we can stop the enumeration immediately as soon as one such solution is found. However, if the number of optimal reconciliations is large and the time-feasible reconciliations are sparse (or nonexistent), this approach would require a long computation time before producing any output.

Based on the work of [Nøj+18], we imagined a method for searching for time-feasible optimal reconciliations that utilizes the enumeration of EL-equivalence classes. Precisely, given an instance, it outputs a time-feasible optimal reconciliation if there is any, or "no" if there is none. In [Nøj+18], the authors presented an algorithm that answers the following question in $O(m \log(n))$ time, where $m = |V(P)|$ and $n = |V(H)|$:

- Given an EL-equivalence class π (see Definition 2.15), is there a reconciliation ϕ in the class π that is time-feasible?

If the answer is yes, it also constructs such a reconciliation ϕ. Our new method is thus straightforward: first we enumerate the EL-equivalent classes, then, for each class, we apply the algorithm of [Nøj+18] to output a time-feasible reconciliation if there is any. There are several pros and cons to this method:

- **Pros:**
  - If there are no time-feasible reconciliations among the optimal ones, with this method we can expect to obtain the "no" answer faster than with the naive method (that is, by enumerating all optimal reconciliations and checking their time-feasibility).
  - If the time-feasible reconciliations are sparse, we can expect to discover one such solution faster.
  - Very often, we want to save the reconciliations to a file for subsequent analysis. In other words, we separate the enumeration and the time-feasibility check into two steps. The space needed for storing the EL-equivalence classes can be much smaller than the space required for storing all optimal reconciliations. The EL-equivalence classes file can be used for the task of finding time-feasible reconciliations as well as other tasks (such as visualization, computation of the average distance or a median solution).
- **Cons:**
  - In an EL-equivalence class, there can be multiple time-feasible reconciliations. This method can output only one of them. It should not be used if we want to find *all* time-feasible optimal reconciliations.

– If the number of EL-equivalence classes is close to the total number of optimal reconciliations (we have seen in Table 3.2 that these two numbers can even be equal), there is little or no advantage in computation time or in storage space compared to the naive method.

As we have seen in Section 3.1.2, for the majority of biological datasets and cost vectors that we have chosen, the number of EL-equivalence classes is quite close to the number of optimal reconciliations, so there is little advantage in terms of the speed or space. Moreover, after enumerating the optimal reconciliations and computing the proportion of those that are time-feasible, we observed that this proportion is almost always high, therefore, we are rarely in a case where the time-feasible reconciliations are sparse or nonexistent. In fact, as we hinted in Chapter 2, Section 2.1.2, in practical situations, by applying the naive method, we are usually able to obtain time-feasible optimal reconciliations.

Nevertheless, there are still a few instances (for example, the dataset COG4964 with the cost vector $(0, 1, 1, 0)$) in which the number of EL-equivalence classes is significantly smaller than the number of optimal reconciliations, and the latter is very large; there are also a few instances (for example, the dataset COG4964 with the cost vector $(0, 2, 3, 1)$) where none of the optimal reconciliations is time-feasible. Take the example of the dataset COG3715 with the cost vector $(-1, 1, 1, 1)$, there are 63360 optimal reconciliations and 2520 EL-equivalence classes (corresponding to a Reduction of 0.04, see Section 3.1.2). The space needed for storing the EL-equivalence classes is thus approximately 0.04 times the space needed for storing all optimal reconciliations, or 25 times smaller. It turned out that none of the optimal reconciliations is time-feasible. We compared the time needed for obtaining the "no" answer using the two methods: (1) apply the $O(n + m)$ time time-feasibility check of [THL11] to the list of optimal reconciliations, (2) apply the $(m \log(n))$ time algorithm of [Nøj+18] to the list of EL-equivalence classes. Roughly, the second method produces the answer 7 times faster than the first one (4 seconds versus 28 seconds). In the future, it is possible that we come across more cases where finding time-feasible reconciliations by means of the EL-equivalence classes would be clearly superior to the naive method.

3.2 Capybara

3.2.1 Overview

Capybara [Wan+20] (see https://capybara-doc.readthedocs.io/) is a cross-platform desktop application that help the user to perform various tasks around the RECONCILIATION PROBLEM. After loading an input file encoding a dataset (H, P, σ), the user can specify a cost vector that makes up the instance of interest, then select one of the following tasks:

- Counting or enumerating all optimal reconciliations.
- Applying the time-feasibility filter during the enumeration of reconciliations.
- Counting or enumerating the V-, E-, and CD-equivalence classes, or enumerating one representative in each equivalence class.
- Enumerating the Best-K reconciliations (see Section 3.2.2).
- Converting a file containing a list of E- or CD-equivalence classes to a format readable by Capybara Viewer, a Web visualization tool that specifically handles these two types of equivalence classes (see Section 3.2.3).

A subset of the features of Capybara is distributed as a Python package, which allows for customized manipulations of the equivalence classes during the enumeration (see Section 3.2.4).

3.2.2 Best-K enumeration

Although at least one time-feasible reconciliations exists for any instance, it does not have necessarily have the minimum cost among all possible reconciliations. When we say that there is no time-feasible optimal reconciliation, it implies that all the time-feasible reconciliations of this instance are *suboptimal*, that is, they have a cost that is strictly larger than the minimum. A natural strategy of searching for time-feasible reconciliations consists, in this case, of enumerating suboptimal reconciliations and then applying the time-feasibility filter.

As early as 1960, finding suboptimal solutions to an optimization problem has been studied for its connection with dynamic programming [BK60; Glu63; WB85]. The Best-K enumeration is an enumeration method that aims at finding a set of k solutions that are *better* than any solution not in this set (they can thus be optimal or suboptimal), and has been used extensively in the problem of finding k shortest paths in a graph [Yen71; KIM82; Epp98; HMS07; AL11]. Intuitively, for an instance of a minimization problem, we can consider a partial order on the set of feasible solutions such that one solution ϕ_1 precedes another solution ϕ_2 if and only if the cost of ϕ_1 is strictly smaller than the cost of ϕ_2. The set of feasible solutions can then be viewed as an ordered list, where the order between solutions of the same cost are chosen arbitrarily. A Best-K enumeration algorithm should output the first K elements of this list, or, if K is larger than the number of feasible solutions, all elements of this list.

Capybara offers the option of Best-K enumeration and its combination with the time-feasibility filter. The algorithm is based on a simple modification of the dynamic programming algorithm for enumerating all optimal reconciliations and was discussed in my Master thesis. It should be noticed that, whether the user chooses the optimal or the Best-K enumeration options, Capybara will check the time-feasibility criteria of [Sto+12], which are more restrictive than the criteria of [THL11].

3.2.3 Visualization of E- and CD-equivalence classes

Given an E-equivalence class, we have the knowledge of the *color* for each parasite node $p \in V(P)$ (see Definitions 2.14 and 2.15). The color is a couple (p, e) consisting of the node p itself and an event type $e \in \mathcal{E} := \{\mathbb{T}, \mathbb{C}, \mathbb{D}, \mathbb{S}\}$ (leaf or terminal event, cospeciation, duplication, host-switch). A natural way of visualizing the E-equivalence classes is to draw the parasite trees with colors on the nodes that indicate the event type. For example, the six reconciliations represented in Figure 2.1 split into four E-equivalence classes; these E-equivalence classes are shown in Figure 3.2.

Using the same color code as Figure 3.2, Capybara Viewer converts a list of E-equivalence classes to a list of pictures that can be viewed in a web browser or downloaded as image files. Each E-equivalence class corresponds to a *frame* that can be displayed by Capy-

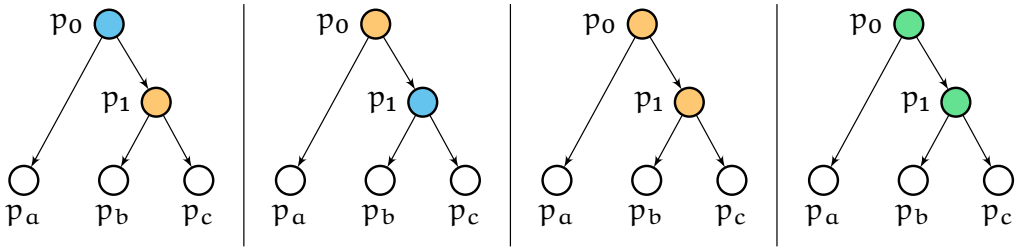


Figure 3.2: Visualization of the four E-equivalence classes of the six reconciliations from Figure 2.1. The colors corresponding to $\mathbb{T}, \mathbb{C}, \mathbb{D}, \mathbb{S}$ event types are respectively white, orange, green, and blue.

bara Viewer; the user can either visualize one particular frame by selecting its index, or visualize all frames sequentially as an animation. Common interaction options such as zooming, panning, and scaling are also available.

For the CD-equivalence classes, we have the additional knowledge of the parasite/host associations, for each parasite node with an event type other than host-switch. Using the same frames as in the visualization of E-equivalence classes, Capybara Viewer displays the label of the host node when the mouse hovers over a parasite node (if the parasite node is of event type host-switch, the host label is the symbol ?, in conformity with our Definition 2.14).

3.2.4 Python package

As we have seen in Section 3.1, counting or enumerating equivalence classes can be a useful method for obtaining biological information from an instance. If we want to include the analysis of equivalence classes as one step of an experimental pipeline, Capybara's Python package can then be a preferred choice over the graphical version of Capybara. The Python package is called `capybara-cophylogeny` and can be found in the repository Python Package Index (PyPI). It is easy to install and to update the package to the newest version using `pip`, the standard Python package manager. The documentation page provides several examples illustrating how to effectively use the package for counting and enumerating equivalence classes in practical situations, such as performing the same task on a large number of instances using multiprocessing, or speeding up the computation using an alternative Python interpreter.

Generally, while the graphical version offers more features and is more suitable for users who are not technically oriented, the Python package offers more flexibility for running multiple tasks automatically and simultaneously.

3.3 Conclusion and perspectives

In Chapter 2, we introduced the RECONCILIATION PROBLEM, defined equivalence relations on reconciliations, and proposed algorithms for counting and enumerating equivalence classes. In particular, we are able to enumerate, in polynomial delay, one representative reconciliation from each equivalence class. In Chapter 3, we applied our algorithms to real datasets from the literature and analyzed their space of optimal reconciliations using equivalence classes. Even when the number of optimal reconciliations is huge (e.g., 10^{42}), we managed to obtain useful information about the solution space through the list of equivalence classes. Finally, we presented Capybara, a user-friendly software which performs various computational tasks based on our algorithms and provides a tool for visualizing equivalence classes.

As a future direction, we would like to extend our algorithms to other biologically motivated definitions of equivalence relations. A future version of Capybara will include more visualization options. How to efficiently visualize several equivalence classes together in a compact and static fashion (i.e., without animation) is an especially interesting and challenging question.

The problem of efficiently enumerating equivalence classes is interesting in its own right. In Chapter 4, we will see a generalization of Algorithm 1 to solve the EQUIVALENCE CLASSES ENUMERATION PROBLEM for general ad-AND/OR graphs; more perspectives will be presented there in the scope of enumeration algorithms.

Chapter 4

Equivalence classes of solutions and ad-AND/OR graphs

4.1 Introduction

An *AND/OR graph* is a well-known structure in the field of Logic and Artificial Intelligence (AI) that represents problem solving and problem decomposition (see, for example [MM78; Nil82]). In this chapter, we consider a particular flavor of AND/OR graphs: the acyclic decomposable AND/OR graphs or ad-AND/OR graphs, also known as the *explicit AND/OR graphs for trees* [DM07]. In Chapter 2, the same structure served as the basis for describing the reconciliation graph, a graph that represents the solution space of an instance of the RECONCILIATION PROBLEM. More generally, ad-AND/OR graphs have an intimate relationship with problems that can be solved by dynamic programming (DP-problems).

As we suggested at the end of Chapter 3, enumerating equivalence classes is an interesting and challenging topic in the field of enumeration algorithms. Unlike the other chapters of this thesis which all have to do with phylogenetic trees, the current chapter is concerned with the problem of enumerating equivalence classes of solutions in the context of DP-problems. The algorithm that we will propose can be seen as a generalization of Algorithm 1 of Chapter 2; this generalization is made from a partic-

ular DP-problem (the RECONCILIATION PROBLEM) to a whole family of DP-problems. Generalizing an algorithm to a wider range of problems is not a trivial task. It is crucial to have a perspicuous formulation of the generalized version of the equivalence classes enumeration problem. To this end, we conveniently use the framework of ad-AND/OR graphs to represent the common structure of those DP-problems, and we define a specific kind of node-coloring on ad-AND/OR graphs, with the aim of characterizing the equivalence relations that we are allowed to consider on the solutions of the DP-problems.

The remainder of the chapter is organized as follows. After presenting the background and the motivation, we will first define an intermediate problem, namely the COLORED CLASSES ENUMERATION PROBLEM on ad-AND/OR graphs, and give a polynomial delay algorithm. Then, as a second step, we show the connection between the colored classes of an ad-AND/OR graph and the equivalence classes of solutions of a DP-problem, and provide several concrete examples to illustrate how our algorithm can be applied to a wide variety of DP-problems.

4.2 Background and motivation

Whereas classical optimization problems require to find a single best solution, enumeration problems require to build an entire set of solutions. The study of enumeration problems offers many theoretical challenges as well as perspectives in practical applications. On one hand, enumeration problems can give rise to theoretical questions that are not applicable in the more traditional setting of decision or optimization problems, such as the output-sensitive complexity analysis (see Section 1.2.3). For instance, the enumeration of minimal traversals of a hypergraph is a major problem that is not known to be solved in polynomial total time [FGS19]. On the other hand, enumeration problems often provide more satisfactory answers to real-world questions compared to optimization problems, as mathematical formulations (such as the sum of weights) may not adequately capture all the desired properties of the best solution. This can be due to various reasons: the quality criteria may be complicated (for example, including the time-feasibility criterion in the RECONCILIATION PROBLEM would

yield a NP-hard problem); the choice may be determined by data that is not yet available (for example, finding a shortest path in a transportation network subject to real time disturbances); the decision may involve subjective factors that cannot be formally modeled. In these situations, an enumeration algorithm is able to provide a list of candidate solutions (optimal, or of high quality), so that the user can apply more complicated criteria, wait for data to become available or make full use of human expertise before making a choice.

While real-world applications motivate the formulation of enumeration problems, the practical usability of enumeration algorithms can be limited. The main reason for this is the large number of solutions. The set of candidate solutions can be too slow to be analyzed by the user or requires complex data analysis methods. One way of dealing with this problem is the generation of representative solutions; similar formulations include the generation of solutions that are far apart form each other, or the generation of a subset of solutions of high diversity. In theses kinds of approaches, we need to carefully define a new problem to better answer the original enumeration problem. Regardless of the meaning of representative solutions, notice immediately that, for the method to be applicable, one either requires the set of candidate solutions to be small enough to be explored to a sufficient extend, or, when that is not the case (for example, when there are candidate 10^{42} solutions), one needs to find the representative solutions directly from the original enumeration problem instance.

In our work, we chose to find representative solutions through the enumeration of equivalence classes; this is by no means the only method of defining and obtaining representative solutions. Depending on the data types, it can be more straightforward to define an equivalence relation rather than a distance measure or a diversity measure on the set of solutions (or the other way around, for example, when the data consists of vectors of numerical values). Mathematically, a distance or diversity measure can be more costly or complicated to compute, as its computation typically involves multiple objects that need to be treated together, whereas the computation of equivalence classes is usually done by mapping each single object to a canonical object of the class to which it belongs.

Although there are similarities in the rationale between methods based on equivalence relations and methods based on distance measures, there are important differences that should not be ignored, especially when interpreting of the results. Along with a chosen distance or diversity measure, the user usually also needs to specify a number k as the size of the output (or an upper bound of the size), in such a way that the level of detail or the granularity of the representative solutions in the output can be explicitly controlled. The number of equivalence classes is directly determined by the definition of the equivalence relation, and consequently, the choices of equivalence relations are limited in practice to those that do not yield too much or too little equivalence classes. Another potential issue is that, in the enumeration output, equivalence classes may be presented without any good way of assessing their relative disparities (two solutions are simply non equivalent to each other, but cannot be "more or less" non equivalent). When the size of each equivalence class (or the fraction of the solution space that it represents) is of importance to the user, this information should also be included in the output to prevent any cognitive bias. Indeed, according to the applications, the user may be more interested in solutions that are the most representative, in the sense that they are similar to a large portion of the whole set of solutions, or she may be more interested in solutions that are outliers. Notice that, in data analysis methods based on distance measures, the preference between community detection and anomaly detection is usually directed encoded in the design of the algorithm. An algorithm that generates a list of equivalence classes does not necessarily need to consider the size of each equivalence class. Later in the chapter, we solve the problem of enumerating equivalence classes and the problem of finding the size of an equivalence class in our framework using two entirely different algorithms.

Computational problems that explore the idea of equivalence relation or equivalence classes have been identified in various areas, such as Genome rearrangements [Bra+08; BS10], Artificial intelligence [AMP97], Pattern matching [Blu+87; Nar+07], or the study of RNA shapes [GVR04]. The efficient enumeration of representative solutions to a problem was listed as an important open problem in a Dagstuhl workshop in 2019 on *Algorithmic Enumeration: Output-sensitive, Input-Sensitive, Parameterized,*

Approximative [FGS19]. This challenge has been addressed in the literature in a few specific problems and using different approaches, such as diversity measures [AK11], super-solutions [MSS14], and equivalence classes [Bra+08; Mor15]. To the best of our knowledge, no generalized method for finding representative solutions has been designed to deal with a family of problems.

4.3 Colored classes of solution subtrees of an ad-AND/OR graph

4.3.1 Basic definitions

First, recall the following notations. If w is a node in a directed graph, its out-neighbors are called its children and the set of children is denoted by $ch(w)$. The root node of a rooted tree T is denoted by $r(T)$. Next, let us restate the definitions of ad-AND/OR graphs given in Chapter 2.

Definition 4.1 (ad-AND/OR graph). A directed graph G is an *acyclic decomposable AND/OR graph* (shortly, ad-AND/OR graph) if it satisfies the following:

- G is a acyclic (it is a DAG).
- G is bipartite: its node set $V(G)$ can be partitioned into $(\mathcal{A}, \mathcal{O})$ so that all arcs of G are between these two sets. Nodes in $\mathcal{A}$ are called *AND nodes*; nodes in $\mathcal{O}$ are called *OR$^+$ nodes*.
- Every AND node has in-degree at least one and out-degree at least one. The set of nodes of out-degree zero is then a subset of $\mathcal{O}$ and is called the set of *goal nodes*; the remaining OR$^+$ nodes are simply the *OR nodes*. The subset of OR nodes of in-degree zero is the set of *start nodes*.
- G is *decomposable*: for any AND node, the sets of nodes that are reachable from each one of its child nodes are pairwise disjoint.

An example of an ad-AND/OR graph is given in Figure 4.1.

Definition 4.2 (Solution subtree). A *solution subtree* T of an ad-AND/OR graph G is a subgraph of G which: (1) contains exactly one start node; (2) for any OR node in T it

contains one of its child nodes in G, and for any AND node in T it contains all its children in G.

The set of solution subtrees of G is denoted by $\mathcal{T}(G)$. It is immediate to see that a solution subtree is indeed a subtree of G: it is a rooted tree, the root of which is a start node. If we would drop the requirement of G being decomposable, the object defined in Definition 4.2 would not be guaranteed to be a tree (in general AND/OR graphs, the solutions are not called *subtrees* but instead *subgraphs*). In Figure 4.1, one solution subtree is shown in bold.

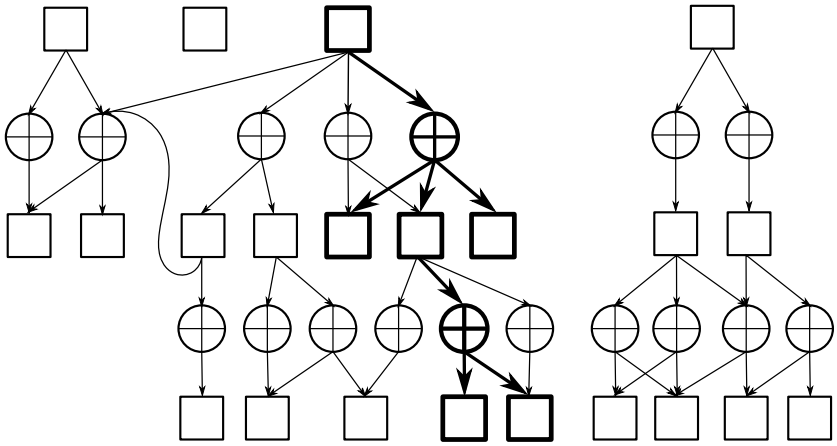


Figure 4.1: An acyclic decomposable AND/OR graph with four start nodes. Squares are OR^+ nodes (OR nodes or goal nodes); crossed circles are AND nodes. One solution subtree of size 8 is shown in bold.

The following four-step procedure allows to obtain one solution subtree: (1) start at any start node, (2) for any visited OR node, visit one child, (3) for any visited AND node, visit all children, (4) stop when the goal nodes are reached. The set $\mathcal{T}(G)$ of solution subtrees of any ad-AND/OR graph G can be enumerated efficiently: after a preprocessing step in time linear in the size of G, the delay between outputting two consecutive solutions is linear in the size of the solution. This result is folklore and the algorithm is sometimes redesigned to fit the needs of a particular problem. Indeed, for a specific dynamic programming problem, the enumeration of the solutions can be done in an *ad hoc* manner, and the graph structure used by the algorithm is not explicitly called an ad-AND/OR graph. For instance, in [Don+15], the authors devised

an enumeration algorithm for the RECONCILIATION PROBLEM that works not only on reconciliation graphs but on any ad-AND/OR graph. See [Rot19] for another example. In the literature that explicitly studies general AND/OR graphs, the effort is naturally focused on the question of enumerating solution subgraphs in AND/OR graphs that are neither acyclic nor decomposable, which is much more challenging (see, e.g., [MM78; MB85; JT00; Gho+12]).

We will define now a notion of subgraph for ad-AND/OR graphs that we will be used extensively. Intuitively, the relationship between such a subgraph and the entire graph correspond to the relationship between a DP-subproblem and the "full" optimization problem.

Definition 4.3 (Subgraph starting from a set of nodes). Let G be an ad-AND/OR graph. Let $\mathcal{O}$ be a set of OR$^+$ nodes of G. The *subgraph of* G *starting from* $\mathcal{O}$, denoted by $G/\mathcal{O}$, is the subgraph obtained from G by setting $\mathcal{O}$ as the new set of start nodes (i.e., by removing all nodes are *unreachable* from $\mathcal{O}$ through directed paths).

When the set $\mathcal{O}$ consists of a single node w, the notation $G/\mathcal{O}$ simplifies to G/w.

It may seem trivial but let us mention that the number of solution subtrees $|\mathcal{T}(G)|$ of an ad-AND/OR graph G can be computed easily, without enumerating any of them. To see this, let $N(w)$ denote the number of solution subtrees of G/w for each OR$^+$ node w. If w is a goal node, then $N(w) = 1$. Otherwise, $N(w) = \sum_{v_i \in ch(w)} \prod_{w_k \in ch(v_i)} N(w_k)$. In a nutshell, the number of solution subtrees of a subgraph starting from an OR and (respectively, an AND node) node is the sum (respectively, the product) of that of its children.

4.3.2 Color classes: definitions

Let G be an ad-AND/OR graph. Let $\mathbb{C}$ be an **ordered** set of colors. The colors will be used to express the equivalence relation on the set of solution subtrees of G. Intuitively, two OR$^+$ nodes having the same color represent two alternative ways of solving the problem that can be considered equivalent.

Definition 4.4 (e-coloring). An ad-AND/OR graph G is *e-colored* if its OR$^+$ nodes are colored in such a way that for any AND node all its children have distinct colors.

Notice here that only OR^+ nodes are colored and AND nodes are not colored. If w is a OR^+node of G, we denote by $c(w)$ its color. If w is an AND node, we denote by $\overline{c}(w)$ the tuple of colors of the children of w sorted in increasing order of the colors. From the definition of an e-coloring, the color tuple $\overline{c}(w)$ of an AND node w is necessarily of the same size as the set of children $\boldsymbol{ch}(w)$. The intuition is that the color tuple $\overline{c}(w)$ distinguishes each child OR^+ node of an AND node w, and gives a specific ordering between them.

If $T \in \mathcal{T}(G)$ is a solution subtree of G, we use the notation $\pi(T)$ for the result of contracting the AND nodes in T: for each OR node w of T, contract the only child node of w in T (i.e., remove the child and connect w to each one of its "grandchildren").

Definition 4.5 (Color class). A node-colored rooted tree C is a *color class* of an e-colored ad-AND/OR graph G if there exists a solution subtree T of G such that $\pi(T)$ is equal to C. Such a T is said to be a solution subtree *belonging to* the color class C.

We denote by $\mathcal{C}(G)$ the set of color classes of G (in Chapter 2, in the context of a reconciliation graph G, the similar object was denoted by $\pi(\mathcal{T}(G))$). The notation π can be seen as a function $\pi\colon \mathcal{T}(G) \to \mathcal{C}(G)$. If C is a color class of G, we denote by $\pi^{-1}(C) := \{T \in \mathcal{T}(G) \mid \pi(T) = C\}$ the subset of solution subtrees of G belonging to the color class C. The notations $c(w)$ and $\overline{c}(w)$ are naturally extended to the case where w is a node in a color class: $c(w)$ denotes the color of w, and $\overline{c}(w)$ denotes the color tuple of the children of w. The color tuple $\overline{c}(\boldsymbol{r}(C))$ of the root of a color class C is called its *root color tuple*.

An example of an e-colored ad-AND/OR graph with five color classes is given in Figure 4.2.

4.3.3 Color classes: enumeration

Given an ad-AND/OR graph G that is e-colored with a fixed ordered color set $\mathbb{C}$, we propose a polynomial delay algorithm to enumerate all equivalence classes of G.

Total ordering over the set of color classes

We define a total ordering $\prec$ (called *smaller than*) over $\mathcal{C}(G)$, the set of color classes of G, using the given ordering on the set $\mathbb{C}$ of colors. If C and C' are two color classes with

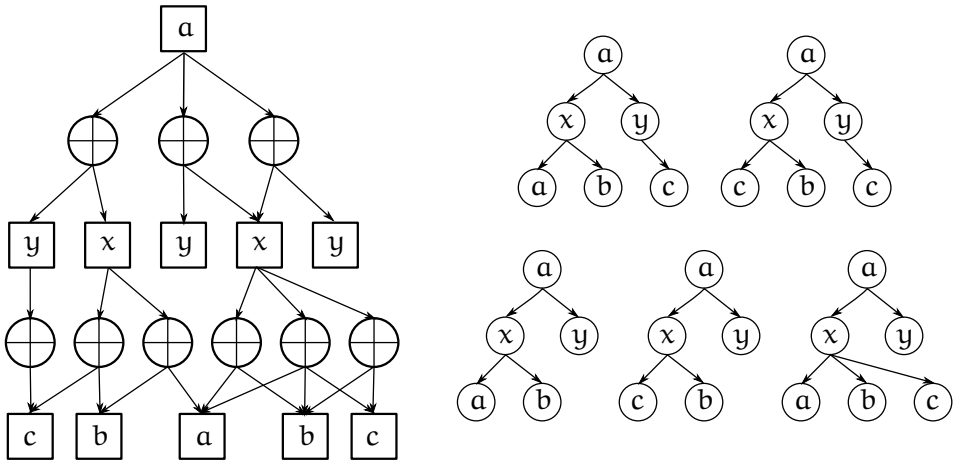


Figure 4.2: An e-colored ad-AND/OR graph and its five color classes. Crossed circles are AND nodes; squares are OR$^+$ nodes. The colors of the OR$^+$ nodes are written inside the squares.

different root colors, we say that C is smaller than C', denoted by $C \prec C'$, if the root color of C precedes that of C'. If C and C' have the same root color, let $(C_1, \dots, C_k)$ (resp. $(C'_1, \dots, C'_\ell)$) be the child subtrees of $r(C)$ (resp. of $r(C')$) sorted recursively with respect to $\prec$. We then say that C is smaller than C' if the tuple $(C_1, \dots, C_k)$ is lexicographically smaller than $(C'_1, \dots, C'_\ell)$, i.e., if $C_i \prec C'_i$ with i being the smallest index such that $C_i \neq C'_i$. We also assume that $\emptyset$ is smaller than any tree, and therefore a single-node tree colored with a color $c \in \mathbb{C}$ comes before any other tree whose root is colored with c in $\prec$.

Definitions and notations

Given an OR node w, we denote by $ct(w)$ the set of color tuples of its children, i.e.,

$$ct(w) := \left\{\overline{c}(v) : v \in ch(w)\right\} .$$

From the definition, a color tuple $(c_1, \dots, c_j)$ belongs to $ct(w)$ if w has a child AND node whose children are colored with $(c_1, \dots, c_j)$. If we consider a color class C of $\mathcal{C}(G/w)$, the tuples in $ct(w)$ are precisely the possible colorings of the children of $r(C)$ (in other words, the root colors of the child subtrees of $r(C)$). Indeed, if the child AND node $v \in ch(w)$ is chosen in a solution subtree, then $\overline{c}(v)$ will be the colors of the children of w in

the color class in which that solution subtree belongs. Notice that several child AND nodes of w may have the same color tuple.

If w is a goal node, we define $\mathit{ct}(w) := \emptyset$. We extend the notation ct to a set $\mathcal{O}$ of OR$^+$ nodes with $\mathit{ct}(\mathcal{O}) := \bigcup_{w \in \mathcal{O}} \mathit{ct}(w)$. In the same way, $(c_1, \ldots, c_j)$ is a color tuple of $\mathit{ct}(\mathcal{O})$ if and only if there exists a color class C of $\mathcal{C}(G/\mathcal{O})$ such that the children of $\mathit{r}(C)$ are colored with $(c_1, \ldots, c_j)$.

Given a set $\mathcal{O}$ of OR$^+$ nodes, we consider the set $\mathit{ct}(\mathcal{O}$ in lexicographical order. For an index $r \leq |\mathit{ct}(\mathcal{O})|$, let t_r be the r-th color tuple of $\mathit{ct}(\mathcal{O})$. We denote by $P^r(\mathcal{O})$ the subset of child AND nodes of $\mathcal{O}$ whose color tuple is t_r, i.e.,

$$P^r(\mathcal{O}) := \{v \in \mathit{ch}(\mathcal{O}) : \overline{c}(v) = t_r\}.$$

When the set $\mathcal{O}$ is clear from the context, it is omitted from the notation and we simply write P^r (in Algorithm 3, we will only work with a fixed set $\mathcal{O}$). The subsets $P^1, \ldots, P^{|\mathit{ct}(\mathcal{O}|}$ form a partition of $\mathit{ch}(\mathcal{O})$.

Finally, given a color tuple $t_r := (c_1, \ldots, c_j) \in \mathit{ct}(\mathcal{O})$, for each $i \leq j$, by Definition 4.4, each AND node in P^r has exactly one child OR$^+$ node colored with c_i. We denote by P^r_i the set of child OR$^+$ nodes of P^r colored with c_i, i.e.,

$$P^r_i = \{w \in \mathit{ch}(v) : v \in P^r,\ c(w) = c_i\}.$$

In summary, the key notation P^r_i that we will use in Algorithm 3 can be described as follows: it is the set of grandchild OR$^+$ nodes of $\mathcal{O}$ colored with the i-th color c_r of the r-th color tuple $t_r \in \mathit{ct}(\mathcal{O})$.

In the left panel of Figure 4.3, an example graph is shown where each node is labeled by an integer. The colors are, in increasing order, w, x, y, and z. For $\mathcal{O} = \{1, 2\}$, the set $\mathit{ct}(\mathcal{O})$ contains the three color tuples $t_1 = (w, y)$, $t_2 = (x, y)$ and $t_3 = (x, y, z)$. The three corresponding sets of child AND nodes are, respectively, $P^1 = \{6\}$, $P^2 = \{4, 5\}$, and $P^3 = \{3\}$. The sets P^r_i are the following: $P^1_1 = \{12\}$, $P^1_2 = \{11\}$, $P^2_1 = \{9, 10\}$, $P^2_2 = \{8, 11\}$, $P^3_1 = \{9\}$, $P^3_2 = \{8\}$, and $P^3_2 = \{7\}$.

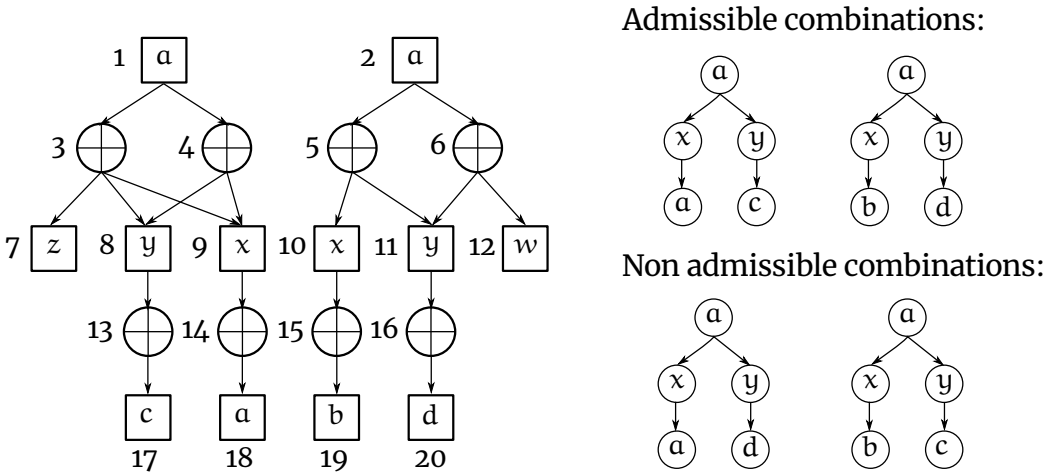


Figure 4.3: Left panel: An e-colored ad-AND/OR graph, where each node is labeled by an integer. Right panel: For $\mathcal{O} = \{1, 2\}$ and $t_2 = (x, y)$, we have $P_1^2 = \{9, 10\}$ and $P_2^2 = \{8, 11\}$. There are four combinations between $\mathcal{C}(G/P_1^2)$ and $\mathcal{C}(G/P_2^2)$; only two of them are admissible.

Algorithm description

In Algorithm 3, we present a function `Next` that, given a color class $C \in \mathcal{C}(G)$, a color $c \in \mathbb{C}$, and and a set of OR^+ nodes $\mathcal{O}$ of G all colored with c, outputs the next color class of $G/\mathcal{O}$ with respect to $\prec$, or, if C is the last one, outputs a special symbol $\perp$. In order to use this function to obtain all color classes, one needs to first partition the set of start nodes of G according to their colors, then, for each color c_k and the subset $\mathcal{O}_k$ of start nodes colored with c_k in increasing order, call `Next` repeatedly with the parameters c_k and $\mathcal{O}_k$, together with each outputted color class C (starting with $\emptyset$) until reaching the symbol $\perp$ (see also the proof of Theorem 4.1).

After treating the easy case where the output color classes consists of a single node (Lines 4–6), Algorithm 3 distinguishes two cases. Since the output must follow the lexicographic order on the set $\boldsymbol{ct}(\mathcal{O})$ of color tuples, the algorithm identifies the index r that corresponds to the root color tuple $\overline{c}(\boldsymbol{r}(C))$ of the current color class C (Line 17), and makes use of the following observation. Either (1) C is the last color class of $G/\mathcal{O}$ with the r-th root color tuple, and in this case we try to move on to the next color tuple (Line 20, the "go to" case), or (2) there still exists some color classes with the same root color tuple.

Algorithm 3: Enumerate the next color class with respect to $\prec$

Input: A color $c \in \mathbb{C}$, a set of OR^+ nodes $\mathcal{O}$ of G all colored with c, and a color class $C \in \mathcal{C}(G/\mathcal{O})$

Output: The color class C' of $G/\mathcal{O}$ that follows C with respect to the $\prec$ ordering, or $\perp$ if no such C' exists.

Function `Next`$(c, \mathcal{O}, C)$**:**

- **if** $C = \emptyset$ *and* $\mathcal{O}$ *contains goal nodes* **then**
  - **Return** A tree with a single root node colored with c
- **end**
- $r \leftarrow 0$
- **if** $C = \emptyset$ *or* C *is a single-node tree* **then**
  - $r \leftarrow r + 1$
  - **if** $r > |\boldsymbol{ct}(\mathcal{O})|$ **then**
    - **Return** $\perp$
  - **end**
  - Let $(c_1, \ldots, c_j)$ be the color tuple of $t_r \in \boldsymbol{ct}(\mathcal{O})$ and let $C_1, \ldots, C_j \leftarrow \emptyset$
  - $\mathcal{O}_1 \leftarrow P_1^r$
  - $\ell \leftarrow 1$
- **else**
  - Let r be such that $t_r \in \boldsymbol{ct}(\mathcal{O})$ is the root color tuple $\overline{c}(\boldsymbol{r}(C))$ of C
  - Let $(C_1, \ldots, C_j)$ be the child subtrees of $\boldsymbol{r}(C)$, the roots of which are colored respectively with $(c_1, \ldots, c_j) := t_r$
  - For all $i \leq j$, let $\mathcal{O}_i \subseteq P_i^r$ be the set of nodes in P_i^r compatible with $(C_1, \ldots, C_{i-1})$, the first $(i-1)$ child subtrees of $\boldsymbol{r}(C)$
  - Let ℓ be the largest index $i \leq j$ such that `Next`$(c_i, \mathcal{O}_i, C_i) \neq \perp$ if such index exists. Otherwise, $C \leftarrow \emptyset$ and go to line 8
- **end**
- $C_\ell \leftarrow$ `Next`$(c_\ell, \mathcal{O}_\ell, C_\ell)$
- **for** $\ell < i \leq j$ **do**
  - Let $\mathcal{O}_i \subseteq P_i^r$ be the set of nodes in P_i^r compatible with $(C_1, \ldots, C_{i-1})$
  - $C_i \leftarrow$ `Next`$(c_i, \mathcal{O}_i, \emptyset)$
- **end**
- **Return** A tree with root color c and root child subtrees $(C_1, \ldots, C_j)$

In this second case, we turn our attention to the child subtrees $(C_1, \dots, C_j)$ of $r(C)$, ordered according to their root colors. Observe that, since C is a color class of $G/\mathcal{O}$, for each $i \leq j$, the i-th child subtree C_i is a color class of G/P_i^r. It then suffices to replace the last child subtree C_j by its successor C'_j if there exists one, and we obtain a solution C' whose child subtrees at the root are $(C_1, \dots, C_{j-1}, C'_j)$, which is by definition of the successor of C. If C_j has no successor, we replace if possible C_{j-1} by its successor C'_{j-1} and we replace $C - j$ by the smallest admissible color class (i.e., the successor of $\emptyset$). In general, we select at each step the greatest index ℓ such that C_ℓ has a successor with respect to $\prec$ (Line 20), and we replace it by its successor C'_ℓ (Line 22), and we take the smallest admissible color class for every $\ell < i \leq j$ (Lines 23-26).

Without further care, the above described procedure would output color classes C' whose child subtrees $(C'_1, \dots, C'_j)$ at the root correspond to the elements of the Cartesian product of $\mathcal{C}(G/P_i^r)$, for $i \leq j$. However, while it is true that if C is a color class of $G/\mathcal{O}$, its child subtree C_i is a color class of G/P_i^r for all $i \leq j$, the converse is not true. Indeed, not all elements of $\mathcal{C}(G/P_1^r) \times \cdots \times \mathcal{C}(G/P_j^r)$ lead to an admissible color class (an example is given in the right panel of Figure 4.3). In order to find an admissible color class, we should guarantee that the choice of a given C_i is *compatible* with the previous choices $(C_1, \dots, C_{i-1})$. This is done by selecting the subset of OR^+ nodes $\mathcal{O}_i \subseteq P_i^r$ that are compatible with $(C_1, \dots, C_{i-1})$ (Lines 19 and 24). An admissible choice of C_i will then be any color class of $G/\mathcal{O}_i$. The definition of compatible nodes will be given later. There are two important remarks with respect to the addition of this compatibility check in the algorithm. First, for each $i \leq j$, the subset $\mathcal{O}_i$ can be easily computed (this can be seen from Definition 4.6). Secondly, $\mathcal{O}_i$ is never empty, i.e., there is always a choice for C_i that is compatible with the previous choices of $(C_1, \dots, C_{i-1})$ (there is at least one choice that corresponds to the current color class). Notice that if this property were not satisfied, the algorithm would not have a polynomial delay complexity since we may spend exponential time without reaching any color class. With this property, we are guaranteed to be always able to extend a partial tuple of subtrees $(C_1, \dots, C_i)$ until we reach a complete tuple $(C_1, \dots, C_j)$ that will form an admissible color class.

Compatible nodes

Given a set of OR$^+$ nodes $\mathcal{O}$ all colored with the same color c and a color class C of $\mathcal{C}(G/\mathcal{O})$, we denote by $r(\pi^{-1}(T)) := \{r(T) : T \in \pi^{-1}(C)\}$ the subset of OR$^+$ nodes in $\mathcal{O}$, each one of which is the root of a solution subtree belonging to the color class C. The following definition formalizes the notion of compatible nodes mentioned previously.

Definition 4.6. Let $\mathcal{O}$ be a set of OR$^+$ nodes of color c. Let $t_r =: (c_1, \ldots, c_j) \in \mathit{ct}(\mathcal{O})$ be its r-th color tuple. Let k be an integer such that $1 \leq k < j$. For $i \leq k$, let C_i be a color class in $\mathcal{C}(G/P_i^r)$. We say that a OR$^+$ node $w \in P_{k+1}^r$ is *compatible* with the k-tuple $(C_1, \ldots, C_k)$ if there exists an AND-node $v \in P^r$ that satisfies the following: (1) w is a child of v, (2) for all $i \leq k$, the set $r(\pi^{-1}(C_i))$ contains a child of v.

In Algorithm 3, we extended Definition 4.6 to the case where $k = 0$, and any node in P_1^r is compatible with the empty tuple.

The next lemma, essential to the proof of correctness, states the key property of compatible nodes: choosing a compatible node guarantees that a partial tuple of subtrees can be "safely" extended, without the risk of creating any non-admissible color class, that is, a color class that does not correspond to any solution subtree of G.

Lemma 4.1. *Let $\mathcal{O}$ be a set of OR$^+$ nodes G all colored with c. Let $t_r =: (c_1, \ldots, c_j)$ be the r-th color tuple in $\mathit{ct}(\mathcal{O})$. Let C be a color class of $G/\mathcal{O}$, and let $C_1, \ldots, C_k$, be its first k child subtrees, for some fixed $k < j$, where $C_i \in \mathcal{C}(G/P_i^r)$ for all $i \leq k$. Let $\mathcal{O}_{k+1} \subseteq P_{k+1}^r$ be the set of OR$^+$ nodes compatible with $(C_1, \ldots, C_k)$. Given $C_{k+1} \in \mathcal{C}\left(G/P_{k+1}^r\right)$, there exists a color class $C \in \mathcal{C}(G/\mathcal{O})$ whose first $(k+1)$ child subtrees at the root are precisely $(C_1, \ldots, C_k, C_{k+1})$ if and only if $C_{k+1} \in \mathcal{C}(G/\mathcal{O}_{k+1})$.*

Proof. (*First direction*) Assume that there exists a color class C of $G/\mathcal{O}$ whose first $(k+1)$ child subtrees at the root are $(C_1, \ldots, C_k, C_{k+1})$, and let T be a solution subtree of $G/\mathcal{O}$ such that $\pi(T) = C$. Let v be the unique child AND node of the $r(T)$. Notice that since we have $C_i \in P_i^r$ for all $i \leq k$, the color tuple $\bar{c}(v)$ of v is necessarily the r-th color tuple $t_r \in \mathit{ct}(\mathcal{O})$, and so $v \in P^r$. Let $(w_1, \ldots, w_j)$ be the child OR$^+$ of v in T, ordered in increasing order of their color. For all $i \leq k$ we have $w_i \in r(\pi^{-1}(C_i))$, the node w_{k+1} is thus compatible with $(C_1, \ldots, C_k)$. Therefore, $C_{k+1} \in \mathcal{C}\left(G/\mathcal{O}_{k+1}\right)$ since $C_{k+1} \in \mathcal{C}(G/w_{k+1})$, and $w_{k+1} \in \mathcal{O}_{k+1}$.

(Second direction) Assume now that $C_{k+1} \in \mathcal{C}(G/\mathcal{O}_{k+1})$. Then there exists a compatible OR$^+$ node $w_{k+1} \in \mathcal{O}_{k+1}$ and a solution subtree T_{k+1} of G/w_{k+1} rooted at w_{k+1} that satisfies $\pi(T_{k+1}) = C_{k+1}$. Since $w_{k+1} \in \mathcal{O}_{k+1}$, by the Definition 4.6 of compatible nodes, there exists an AND node $v \in P^r$ such that w_{k+1} is a child of v, and, for all $i \leq k$, there exists an OR$^+$ node $w_i \in r(\pi^{-1}(C_i))$ such that $c(w_i) = c_i$ (i.e., w_i is the unique child of v colored with the color c_i). Therefore, for all $i \leq k$, there exists a solution subtree T_i of G/w_i such that $\pi(T_i) = C_i$. Now consider any solution subtree T of $G/\mathcal{O}$ containing the nodes $v, w_1, \ldots, w_k$, and w_{k+1}. Then its first $(k+1)$ child subtrees at v are $(T_1, \ldots, T_k, T_{k+1})$, and therefore, the first $(k+1)$ child subtrees at the root of its color class $\pi(T)$ are precisely $(C_1, \ldots, C_k, C_{k+1})$. □

Analysis

Lemma 4.2. *Let C be a color class of $G/\mathcal{O}$ for a set $\mathcal{O}$ of OR$^+$ nodes of G, all colored with the same color c. Then, the function `Next` of Algorithm 3 satisfies the following:*

1. *`Next`$(c, \mathcal{O}, \emptyset)$ returns the smallest equivalence class of $G/\mathcal{O}$ with respect to $\prec$.*
2. *If C is not the last color class of $G/\mathcal{O}$, then `Next`$(c, \mathcal{O}, C)$ returns the successor of C.*
3. *If C is the last color class of $G/\mathcal{O}$, then `Next`$(c, \mathcal{O}, C)$ returns $\perp$.*

Proof. Let us define the height of the subgraph $G/\mathcal{O}$, denoted by $h(G/\mathcal{O})$, to be the maximum height of a color class of $G/\mathcal{O}$, i.e., the number of OR$^+$ nodes in a longest path from $\mathcal{O}$ to a goal node minus 1. The proof will be done by induction on $h(G/\mathcal{O})$.

Assume first that $h(G/\mathcal{O}) = 0$, i.e., $\mathcal{O}$ contains only goal nodes. Then $G/\mathcal{O}$ has only one color class C, which is the single-node tree of color c. The call of `Next`$(c, \mathcal{O}, \emptyset)$ will output C at Line 5 of the algorithm, and the call of `Next`$(c, \mathcal{O}, C)$ will return $\perp$ at Line 11 since $|ct(\mathcal{O})| = 0$.

Assume now that $h(G/\mathcal{O}) > 0$.

Proof of Lemma 4.2.1 If $\mathcal{O}$ contains goal nodes, then the smallest color class of $G/\mathcal{O}$ with respect to $\prec$ is the single-node tree colored with c and `Next`$(c, \mathcal{O}, \emptyset)$ outputs it at Line 5. Otherwise, let C' be the smallest color class of $G/\mathcal{O}$ and let $(C'_1, \ldots, C'_j)$ be the its child subtrees at the root. By definition of $\prec$, the roots of $(C'_1, \ldots, C'_j)$ are colored with

the lexicographically smallest color tuple $t_1 := (c_1, \ldots, c_j)$ of $ct(\mathcal{O})$, and by Lemma 4.1, for all $i \leq j$, C'_i is the smallest color class of $G/\mathcal{O}_i$, where $\mathcal{O}_i \subseteq P^1_i$ is the set of nodes of P^1_i compatible with $(C'_1, \ldots, C'_{i-1})$. Thus, $\texttt{Next}(c, \mathcal{O}, \emptyset)$ will return C' at Line 28, since r will receive 1 at Line 9, C_1 will receive $\texttt{Next}(c_1, P^1_1, \emptyset)$ at Line 22 which is equal to C'_1 by the induction hypothesis, and for all $i \leq j$, C_i will receive $\texttt{Next}(c_i, \mathcal{O}_i, \emptyset)$ at Line 25 which is equal to C'_i by the induction hypothesis.

Proof of Lemma 4.2.2 Let $(C'_1, \ldots, C'_j)$ be the child subtrees of $r(C)$ and let $t_r := (c_1, \ldots, c_j)$ be the root color tuple of C, i.e., the color tuple with which the roots of the subtrees $(C'_1, \ldots, C'_j)$ are colored. Let $\mathcal{O}_i \subseteq P^r_i$ be the set of nodes of P^r_i compatible with $(C'_1, \ldots, C'_{i-1})$ for all $i \leq j$. Let C' be the color class of $G/\mathcal{O}$ that follows C with respect to $\prec$. Let $(C''_1, \ldots, C''_j)$ be the child subtrees of $r(C')$. Notice that the children of $r(C')$ are either colored with the same color tuple t_r, or colored with t_{r+1}, the next color tuple of $ct(\mathcal{O})$, if C is the largest color class of $G/\mathcal{O}$ with the root color tuple t_r.

Assume first that the children of $r(C')$ are colored with the same color tuple t_r. Let ℓ be the smallest index such that $C'_\ell \neq C''_\ell$. We claim that ℓ is also the largest index such that C'_ℓ has a successor in $\mathcal{C}(G/\mathcal{O}_\ell)$ with respect to $\prec$, and thus it corresponds to the ℓ chosen by the algorithm at Line 20. Indeed, assume that there exist $i < k \leq j$ and a larger color class $\widehat{C}_k \in \mathcal{C}(G/\mathcal{O}_k)$ such that $C'_k \prec \widehat{C}_k$. By Lemma 4.1, there exists a color class of $G/\mathcal{O}$ whose first k child subtrees at the root are $(C'_1, \ldots, C'_{k-1}, \widehat{C}_k)$. However, in this case such a color class would be greater than C and smaller than C' with respect to $\prec$, and it would be in contradiction with the fact that C' immediately follows C in $\prec$. Now, C_ℓ will receive $\texttt{Next}(c_\ell, \mathcal{O}_\ell, C'_\ell)$ at Line 22, which is by the induction hypothesis the color class that succeeds C'_ℓ in $\mathcal{C}(G/\mathcal{O}_\ell)$. Since we assumed that ℓ is the smallest index such that $C'_\ell \neq C''_\ell$, by definition of $\prec$ and by Lemma 4.1, C''_ℓ is the color class that succeeds C'_ℓ in $\mathbb{C}(G/\mathcal{O})\ell)$, and so C_ℓ will receive C''_ℓ at Line 22. Since for all $i \leq \ell$ we have $C'_i = C''_i$, and since $(C_1, \ldots, C_{\ell-1})$ are not modified by the algorithm, at the end of it, $(C_1, \ldots, C_{\ell-1}, C_\ell)$ will be equal to $(C'_1, \ldots, C'_{\ell-1}, C''_\ell) = (C''_1, \ldots, C''_{\ell-1}, C''_\ell)$. It now remains to show that C_i will be equal to C''_i for all $\ell < i \leq j$. Again, by Lemma 4.1, for all $\ell < i \leq j$, C''_i is the smallest color class of $G/\mathcal{O}'_i)$, where $\mathcal{O}'_i$ is the set of nodes of P^r_i compatible with $(C''_1, \ldots, C''_{i-1})$, since otherwise, another color class of $G/\mathcal{O}$ greater than C and smaller

than C' could be built. Thus, at Line 25, C_i will receive $\texttt{Next}(c_i, \mathcal{O}'_i, \emptyset)$, which is equal to C''_i by the induction hypothesis, and C' will be returned at Line 27.

Assume now that the children of $\boldsymbol{r}(C')$ are colored with the next color tuple t_{r+1}. In this case, C is the greatest color class of $\mathcal{C}(G/\mathcal{O})$ with the root color tuple t_r. We claim that each child subtree C'_i of $\boldsymbol{r}(C)$ is also the greatest color class of $G/\mathcal{O}_i$ for all $i \leq j$. Indeed, assume otherwise that, for some i, there exists a color class $\widehat{C}_i \in \mathcal{O}(G/\mathcal{O}_i)$ with $C'_i \prec \widehat{C}_i$. Then, by Lemma 4.1, there exists a color class $\widehat{C} \in \mathcal{C}(G/\mathcal{O})$ with the same root color tuple t_r that is greater than C, contradicting the fact that C is the greatest color class with the root color tuple t_r. By the induction hypotheses, $\texttt{Next}(c_i, \mathcal{O}_i, C'_i)$ will therefore return $\perp$ for all $i \leq j$. So C will receive $\emptyset$ at Line 20 and the algorithm will go to Line 8, and the next color tuple t_{r+1} will be selected at Line 9. Using now similar arguments to the ones used in the proof of Lemma 4.2.1, the smallest color class with the root color tuple t_{r+1} will be returned.

Proof of Lemma 4.2.3 Assume now that C is the last equivalence class of $G/\mathcal{O}$. Notice that in this case, the children of $\boldsymbol{r}(C)$ are colored with t_r where $r = |\boldsymbol{ct}(\mathcal{O})|$. As previously, let C'_i be the i-th child subtree at $\boldsymbol{r}(C)$. Then, for all $i \leq j := |t_r|$, C'_i is the largest color class of $G/\mathcal{O}_i$. By the induction hypothesis, $\texttt{Next}(c_i, \mathcal{O}_i, C'_i)$ would return $\perp$ for all $i \leq j$. Therefore, C will receive $\emptyset$ at Line 20 and the algorithm will return to Line 8. Since $r = |\boldsymbol{ct}(\mathcal{O})|$, r will receive $r+1$ at Line 9 and $\perp$ will be returned at Line 11. □

Theorem 4.1. *Given an e-colored ad-AND/OR graph G, the set $\mathcal{C}(G)$ can be enumerated with delay $O(n \cdot s)$ where n is the number of nodes of G and s is the maximum size of a color class.*

Proof. To enumerate $\mathcal{C}(G)$, we first partition the set of start nodes of G according to their colors. For each subset $\mathcal{O}_i$ of color c_i, starting with $C := \emptyset$, we repeatedly assign the output of $\texttt{Next}(c_i, \mathcal{O}_i, C)$ to C and output it until $C = \perp$. By Lemma 4.2, this guarantees that we output every color of $\mathcal{O}(G/\mathcal{O}_i)$ exactly once. Since any color class of G belongs to $\mathcal{C}(G/\mathcal{O}_i)$ for a given $i \leq k$, every color class will be outputted exactly once.

For the complexity, notice that at most one recursive call is performed by each node of the next color class. More precisely, if $\texttt{Next}(c, \mathcal{O}, C)$ returns C', there will be exactly one recursive call per node in C' that is not in C, and thus at most s recursive calls will

be performed.

In each recursive call, the set of color tuples $\boldsymbol{ct}(\mathcal{O})$ and the sets P_i^r, where $r \leq |\boldsymbol{ct}(\mathcal{O}|$ and $i \leq j := |t_r|$, can all be computed in $O(n)$ time in total. It remains to show that the sets of compatible nodes $\mathcal{O}_i$, $i \leq j$, can be computed in $O(n)$ time in total which will conclude the proof. To do this, we should be able to compute the sets $\boldsymbol{r}(\pi^{-1}(C_i))$ for all $i \leq j$. If $\texttt{Next}(c, \mathcal{O}, C) = C'$, the easiest way is to return the set $\boldsymbol{r}(\pi^{-1}(C'))$ together with C' when the call $\texttt{Next}(c, \mathcal{O}, C)$ returns. To compute this, observe that if $C \in \mathcal{C}(G/\mathcal{O})$ where $\mathcal{O}$ is a set of goal nodes all having the same color c, then $\boldsymbol{r}(\pi^{-1}(C)) = \mathcal{O}$, and if $\boldsymbol{r}(C)$ has child subtrees $(C_1, \ldots, C_j)$, then $\boldsymbol{r}(\pi^{-1}(C))$ is the set of nodes of $\mathcal{O}$ that has at least one child AND node v such that $\boldsymbol{ch}(v)$ contain exactly one node in $\boldsymbol{r}(\pi^{-1}(C_i))$ for each $i \leq j$, which can be found in $O(n)$ time. Thus only $O(n)$ time is necessary at each recursive call to return $\boldsymbol{r}(\pi^{-1}(C'))$ in addition to C'. □

4.3.4 Restricting the graph to a color class

After the enumeration of the color classes, it might be interesting to go back to the solution subtrees that belong to each color class. In particular, in practical applications, one might want to use the number of solution subtrees as a measure for the "importance" or "significance" of a color class. In Algorithm 4, we present an algorithm that, given an e-colored ad-AND/OR graph G and a color class $C \in \mathcal{C}(G)$, constructs the *subgraph* G^C *of* G *restricted to* C, that is, a subgraph of G of which the solution subtrees are exactly the ones of G belonging to the color class C:

$$\mathcal{T}(G^C) = \pi_G^{-1}(C) := \{T \in \mathcal{T}(G) : \pi(T) = C\}.$$

Once the graph G^C is obtained, the following questions can be answered (by applying the same method as for any ad-AND/OR graph): counting the number of, and enumerating the solution subtrees belonging to the color class C.

Algorithm 4 relies on two recursive functions $\texttt{VisitOR}^+$ and $\texttt{VisitAND}$, both taking as input a node in G and a node in C. The **Require** statements are used to specify the preconditions that the two parameters of the two $\texttt{Visit}$ functions must verify; it can be checked by inspection that these conditions are always satisfied whenever the functions are called. The algorithm performs an operation called **Mark** on the nodes in G.

All nodes are initially unmarked; the Mark operation changes the state of a node into marked.

Recall that $\pi(T)$ transforms a solution subtree $T \in \mathcal{T}(G)$ into a color class $C \in \mathcal{C}(G)$ by contracting the AND nodes in T. For a fixed T, we extend this notation and write $\pi(w) = \pi(v) := x$ for every OR node w in T with its unique AND-child node v in T that are identified with the node x in C under the transformation.

Lemma 4.3. *Let T be a solution subtree of G belonging to the color class C. For every node w of T, there is a call to either the function* `VisitOR`$^+$ *or to* `VisitAND` *of Algorithm 4 with parameters w and $\pi(w)$, depending on whether w is an OR$^+$ node or an AND node.*

Proof. By top-down induction. The start node of T is visited at Line 3 since it has the correct color. In the induction step we separate two cases. For an OR$^+$ node w of T that is not a start node, suppose that the parent v_0 of w in T is visited in a call `VisitAND`$(v_0, \pi(v_0))$. Then w is visited (Line 22), and the second parameter is $\pi(w)$. In the other case, for an AND node v of T, suppose that the parent w_0 of v in T is visited in a call `VisitOR`$^+(w_0, \pi(w_0))$. Since we have $\pi(v) = \pi(v_1)$ and $\overline{c}(\pi(w_0)) = \overline{c}(v)$, the condition at Line 12 is satisfied and v is visited in a call `VisitAND`$(v, \pi(v))$. □

The correctness of Algorithm 4 is shown in Theorem 4.2. We omit the analysis of complexity as the algorithm clearly requires a running time that is linear in the size of the input graph G.

Theorem 4.2. *The set of solution subtrees of G^C, the graph returned by Algorithm 4, is equal to $\pi_G^{-1}(C)$, i.e., the set of solution subtrees of G belonging to the color class T.*

Algorithm 4: Restricting the graph to a color class

Input: *An e-colored ad-AND/OR graph* G *and a color class* C

Output: *The subgraph* G^C *of* G *restricted to* C

Function `Main(G, C)`**:**

  for *each start node* w_0 *of* G *such that* $c(w_0) = c(\boldsymbol{r}(C))$ **do**

    `VisitOR`$^+(w_0, \boldsymbol{r}(C))$

  end

  return G^C obtained from G by removing all unmarked nodes

Function `VisitOR`$^+(w, x)$**:**

  Require: $c(w) = c(x)$

  if w *is a goal node and* x *is a leaf* **then**

    Mark(w)

    return

  end

  for *each child AND node* v_i *of* w *in* G *such that* $\overline{c}(v_i) = \overline{c}(x)$ **do**

    `VisitAND` (v_i, x)

  end

  if *at least one child of* w *is marked* **then**

    Mark(w)

  end

Function `VisitAND`(v, x)**:**

  Require: $\overline{c}(v) = \overline{c}(x)$

  for *each child OR*$^+$ *node* w_i *of* v **do**

    $x_i \leftarrow$ the unique child of x such that $c(w_i) = c(x_i)$

    `VisitOR`$^+$ (w_i, x_i)

  end

  if *all children of* v *are marked* **then**

    Mark(v)

  end

Proof. (First direction) We show that any solution subtree of G^C is also a solution subtree of C, and that it belongs to the color class C. For every marked OR node, at least one child is marked (Line 15); for every marked AND node, all its children are marked (Line 24). A solution subtree of G^C is thus a solution subtree of G. Let T be a solution subtree of G^C, consider the recursion tree of the `Visit` function calls during which the nodes in T are marked. By the preconditions of the `Visit` functions, that is, $c(w) = c(x)$ and $\overline{c}(v) = \overline{C}(x)$, the tree $\pi(C)$ is equal to the tree formed by the colored nodes that are used as the second parameter x in this recursion tree. The latter is simply equal to C (we start with the root of C, then visit each child of the current node), so we have $\pi(T) = C$. Therefore, every solution subtree of G^C belongs to the color class C.

(Second direction) Let T be a solution subtree of G such that $\pi(T) = C$, we show that every node in T is marked by bottom-up induction. By Lemma 4.3, any goal node w in T is visited in a call $\texttt{VisitOR}^+(w, \pi(w))$ so w is marked (Line 9) because $\pi(w)$ is necessarily a leaf. For the induction step we separate two cases. Let w be an OR node in T, and suppose that all nodes in the subtree $T|_w$ are already marked. Again, by Lemma 4.3, w is visited. Then w is marked at Line 16, since exactly one child of w is in T and is thus marked. In the other case, let v be an AND node in T and suppose that all nodes in $T|_v$ are marked. By the lemma, v is visited. Then v is marked at Line 25, because all children of v are in T and are thus marked. This completes the proof. □

4.4 Application to dynamic programming

4.4.1 A formalism for tree-sequential dynamic programming

Since its introduction by Karp and Held [KH67], *monotone sequential decision processes* (mSDP) have been the classical model for problems solvable by dynamic programming (DP-problems). This formalism is based on finite-state automata. The solutions of DP-problems are thus equivalent to languages of regular expressions, or to paths in directed graphs. It is known that Bellman's *principle of optimality* [Bel13] also applies to problems for which the solutions are not sequential but *tree-like* [Bon70]. Various generalizations have been proposed to characterize broader classes of problems solv-

able by DP or DP-like techniques [Hel89; BDI11].

The framework of equivalence classes enumeration that we consider is situated within the immediate generalization of the mSDP model, i.e., generalizing finite automata (regular expressions, paths in DAGs) to finite tree automata (regular tree grammars, solution trees of general AND/OR graphs). Further generalizations exist (from trees to graphs of treewidth > 1); the collection of these methods is known as *Nonserial dynamic programming* [BB72].

In this model, a tree-sequential problem can be specified by a finite (bottom-up) tree automaton $A = (Q, \Sigma, \delta, q_0, Q_F)$, where Q is a finite set of states, Σ is a ranked alphabet, δ is a set of transition rules of the form $(q_1, \ldots, q_n, a, q)$ where $q_1, \ldots, q_n, q \in Q$ and $a \in \Sigma$, $q_0 \in Q$ is the initial state, $Q_F \subseteq Q$ is a set of final states. The problem specification also includes a cost function. The set $L(A)$ of trees accepted by the tree automaton A defines the set of feasible solutions. The minimization problem seeks to minimize the cost function over the set $L(A)$ of feasible solutions.

We will consider the simple case of a positive additive cost function that always equals zero in the initial state. An additive cost function can be defined via an incremental cost function $I\colon Q^* \times \Sigma \to \mathbb{R}$, where Q^* consists of tuples of states in Q of the form $(q_1, \ldots, q_n)$. $I(q_1, \ldots, q_n, a)$ can be viewed as the cost of attaching n child subtrees to a new root of symbol a. While it might seem restrictive to require an additive structure on the cost function, this simple case does cover many important problems admitting a DP-algorithm, for instance, TRAVELLING SALESMAN [Bel62; HK62], KNAPSACK [KPP04], or LEVENSHTEIN DISTANCE [WF74].

In this case, the answer of the minimization problem can be shown to be equal to $\min_{q \in Q_F} D(q)$, where $D : Q \to \mathbb{R}_{\geq 0}$ is defined by the following recurrence equations:

$$\begin{aligned} &D(q_0) = 0\,, \\ &\text{for } q \neq q_0\,, \quad D(q) = \min_{(q_1,\ldots,q_n,a,q)\in\delta} \sum_{1\leq i\leq n} D(q_i) + I(q_1, \ldots, q_n, a)\,. \end{aligned} \tag{4.1}$$

A *dynamic programming algorithm* for the minimization problem corresponds to an algorithm that computes D; the function D is commonly called a *dynamic programming table* (a DP-tabled, also called a DP-array, or a DP-matrix). In general, to be able to write down the recurrence relations does not imply that there exists an efficient algo-

rithm to compute the function. Indeed, such an algorithm does not exist in general for given arbitrary tree automata and cost functions [Iba74].

Using an algebraic approach, Gnesi and Montanari [GMM81] have shown that solving the functional Equation (4.1) corresponds to finding the solution subtrees of a general AND/OR graph. An important special case in which DP-algorithms exist is when the underlying AND/OR graph is acyclic.

When a fixed tree is given as an input to the problem, the underlying AND/OR graph is acyclic and decomposable (that is, it is an ad-AND/OR graph). Such problems are hence naturally solvable by DP-algorithms. These algorithms are known in folklore under the name *Dynamic programming on a tree*. Many graph-theoretical problems (e.g., maximum matching, longest path) can be solved optimally on trees by DP-algorithms. Numerous real-world applications also rely on DP-algorithms on trees; examples can be found, for instance, in Data Science [RM08], Computer Vision [FH05; Vek05], and Computational Biology [BAK12; Don+15].

Explicit construction of the ad-AND/OR graph for DP on a fixed tree

Due to its usefulness for the examples that we will develop next, in the case of DP on a fixed tree, an explicit construction of the ad-AND/OR graph from Equation (4.1) is described below. The construction is done in two steps. In the first step, we build a graph in which every node retains an additional attribute, its *value*, and every OR^+ node is labeled by a state $q \in Q$. In the second step, we *prune* the graph by removing nodes that do not yield optimal values.

1. For each $(q_0, a, q) \in \delta$, create a goal node of value 0 labeled by q. Then, for each $q \neq q_0$ in post-order,

   i. For each $(q_1, \ldots, q_n, a, q) \in \delta$, create an AND node, connect it to the n OR^+ nodes labeled by $q_1, \ldots, q_n$. Its value is equal to the sum of the values of its children, plus $I(q_1, \ldots, q_n, a)$.

   ii. Create a single OR node, connect it to every AND node created in the previous step. Its label is q, and its value is the minimum of the values of its children.

2. For each $q \in Q_F$, remove the OR node labeled by q unless its value is equal to $\min_{q \in Q_F} G(q)$. For each OR node s, remove the arc to its AND-child node s_i if the value of s_i is not equal to the value of s. Finally, remove recursively all AND nodes without incoming arcs.

A similar procedure was described in Chapter 2, Section 2.1.3 for constructing the reconciliation graph, which is an ad-AND/OR graph with some additional properties (one such property is that every AND node has exactly two ordered children).

4.4.2 Examples

We will give several examples of DP-problems for which the enumeration of equivalence classes of solutions can be done by enumerating the color classes of the underlying ad-AND/OR graph.

Optimal tree coloring problem

Description A prototypical problem that fits into the framework of DP on a fixed tree is OPTIMAL TREE COLORING, that is, finding an optimal node-coloring of the input tree. Many problems of practical interest can reduce to OPTIMAL TREE COLORING; three concrete examples are given later in this section.

If T is the input (rooted, ordered) tree and $\mathbb{C}$ is the set of colors, such a problem seeks a coloring $\phi\colon V(T) \to \mathbb{C}$ that minimizes the cost function. There can be many constraints on the coloring function: some nodes of T may be forced to have a certain color, the possible colors of a node may depend on the colors of its descendants. In our tree-sequential dynamic programming formalism, a tree automaton and a cost function are given as part of the input. The tree automaton defines the set $L(A)$ of feasible coloring functions satisfying all those constraints. A state q can be interpreted as a colored subtree of T with a particular root color; the unique initial state is an empty coloring and transitions into a colored leaf of T; a final state corresponds to a fully colored T with a particular root color. A commonly used form of cost functions considers the (possibly weighted) sum over the edges of the tree of the cost of putting two colors on each end of an edge, that is, an incremental cost function I of the form

$I(q_1, \ldots, q_n, a, p) = \sum_{1 \leq i \leq n} p(a_i, a)$ where a_i is the color of the root of the subtree in state q_i and $p\colon C^2 \to \mathbb{R}_{\geq 0}$ is a function that gives the cost of putting two colors at each end of an edge.

Equivalence relations on the set of solutions A possible strategy to define equivalence classes on the solution space of the OPTIMAL TREE COLORING problem is to consider some colors to be locally equivalent on a node. In practical applications, the space of colors can be quite large (this is the case for the RECONCILIATION PROBLEM from Chapter 2). Even though the precise colors of each node are necessary for correctly computing the cost function, when the solutions are analyzed by a human expert, it can be desirable to omit the colors and just look at whether the color of a node belongs to some group of colors. Therefore, this kind of equivalence relations is natural in many situations. Our Definition 4.5 of color classes of an e-colored ad-AND/OR graph deals exactly with equivalence relations of this type.

Let e be a function that maps a node u and a color c to the "color group" of that node, denoted by $e(u, c)$. Two solutions of the OPTIMAL TREE COLORING problem $\phi_1, \phi_2\colon V(T) \to C$ are said to be *equivalent* if $\forall u \in V(T)$, $e(u, \phi_1(u)) = e(u, \phi_2(u))$. Let G be the ad-AND/OR graph associated with this instance. Then G can be e-colored as follows. For each OR$^+$ node w of G labeled with the state q, where q is interpreted as a colored subtree of $T|_u$ of T rooted at a node u colored with c, we color the node w with $e(u, c)$. After this, G is e-colored, and its set of color classes $\mathcal{C}(G)$ corresponds to the set of equivalence classes of the solutions of the instance. Notice that the constraint we had on the e-coloring of an ad-AND/OR graph is naturally satisfied by any meaningful function e because in a DP setting we only consider ordered trees: the i-th and j-th children of a node of T cannot be in the same color group unless $i = j$.

Concrete examples of tree coloring problems

In each of the following problems, the connection with the OPTIMAL TREE COLORING problem is straightforward. The aim here is to show that our framework allows to consider equivalence relations that are natural or appropriate for those problems.

Example 1 In the RECONCILIATION PROBLEM that we studied in Chapter 2, the input tree is the parasite tree, the set of colors is the set of nodes of the host tree, and the notion of reconciliation defines the set of feasible colorings. The E-, EL-, and CD-equivalence relations, all of which can be shown to yield an e-coloring of the reconciliation graph, are motivated by biological considerations and by practical needs. In Chapter 3, we have seen that useful information can be extracted from the equivalence classes to help us better understand the solution space, which can be huge for some datasets.

Example 2 This example is related to the alignment of gene sequences on a phylogenetic tree, known as the TREE ALIGNMENT [San75]. In this problem, one seeks to infer the ancestral sequences, knowing the sequence on each taxa, in a way that is most parsimonious (i.e., minimizing the edit distance between adjacent sequences). The general version of the problem considers sequences of arbitrary length and is NP-hard [WJ94; War95]. Here we look at the special case of sequences of length 1 (which also solves the problem of aligning, on a tree, sequences of a given fixed length). The input is a tree T, a set Σ of letters (DNA alphabet or protein alphabet), a function that labels each leaf node of T with a letter, and a distance function $d\colon \Sigma^2 \to \mathbb{R}_{\geq 0}$ between two letters. The goal is to extend the leaf labeling to a full labeling $\phi\colon V(T) \to \Sigma$ such that the sum of the distances over the edges of T is minimized. Defining equivalence relations of the solutions based on a grouping of the letters is uncontrived in this problem: for the DNA or protein alphabet, the letters can be subdivided into structurally similar groups. In practice, we have a fixed length n for the sequences on each node, and we solve the single-letter problem independently for each of the n positions and then combine the solutions together. It is clear that, by combining n sets of solutions, the number of solutions can quickly become huge when the length grows. Summarizing the solution space by means of enumerating the equivalence classes is then a useful technique for analyzing the output.

Example 3 The FREQUENCY ASSIGNMENT problems are a family of problems that naturally arise in telecommunication networks, and that have been extensively stud-

ied in graph theory as a generalization of graph coloring known as the T-coloring problem [Hal80; Rob91; Tes93]. In the variant called the list T-coloring, the input is a graph G representing the interference between radio stations, a set C of colors, a function S that gives for each vertex $v \in V(G)$ a set $S(v) \subseteq C$ of colors (possible frequencies for a station), and a set $T \subseteq C^2$ of forbidden pairs of colors (interfering frequencies). The goal is to find a coloring $\phi : V(G) \to C$ such that a $\forall v \in V(G)$, $\phi(v) \in S(v)$, and $\forall (u, v) \in E(G)$, $(\phi(u), \phi(v)) \notin T$. While this problem is hard in general, it can be solved by DP when the underlying graph is a tree. In this case, we can enumerate colorings of the input trees without any forbidden pair of colors on the edges. Defining equivalence relations by grouping some of the colors together (for example, frequencies that could be considered similar), and enumerating only one solution per equivalence class can be a practical way of reducing the size of the output.

Graph optimization problems and DP on tree decomposition

Graph optimization problems can model a large variety of real-world problems. Many hard graph problems can be solved in polynomial time with a dynamic programming algorithm when the input graph has bounded treewidth (see, for example, [Bod88]). The underlying idea is that, given a tree decomposition of a graph, the DP-algorithm traverses the nodes (bags) of the decomposition and consecutively solves the respective sub-problems.

A well-studied type of graph problems is the vertex subset optimization problems, which ask for a subset of vertices of the input graph that is optimal with respect to some graph-theoretical properties. Given a bag X, a DP-algorithm generally computes for each subset $Z \subseteq X$ the optimal solution of the DP-subproblem whose intersection with X is Z. In this context, we could define two solutions to be equivalent if they intersect each bag of the decomposition in an "equivalent" way. The equivalence relation on the solutions is then defined by an equivalence relation over the subsets of each bag, and two solutions S_1 and S_2 are equivalent if for all bags X of the decomposition, $S_1 \cap X$ is equivalent to $S_2 \cap X$.

One of the simplest examples is to consider that all the nonempty subsets of ver-

tices of a bag are equivalent. Thus, what we are interested in is whether a solution "hits" a bag (i.e., whether it has a nonempty intersection with the vertices in the bag). Consequently, two solutions would be considered equivalent if they hit the same bags.

We can also consider two subsets of a bag to be equivalent if they have the same size. In this case, two solutions would be equivalent if each bag contains the same number of vertices in the two solutions.

The above two examples give an idea on how equivalence relations which fit into our framework can be defined. We believe that considering solutions in the way they are distributed along the tree decomposition of a graph could give a good overview of the diversity of the solution space.

4.5 Conclusion and perspectives

In this chapter, we provided a framework for the enumeration of solutions in polynomial delay that can be applied to a whole family of problems, and we showed several examples of applications in dynamic programming problems. A few key ingredients are required: one is that the solutions must be represented in an AND/OR graph that is acyclic and decomposable, the other is that the equivalence relation must be encoded into an e-coloring of the graph. It would be interesting to ask the same question for a general colored AND/OR graph, that is, whether we can efficiently enumerate color classes of solution subgraphs, if one or more of those ingredients is missing. In particular, it remains open whether the problem is hard without assuming the decomposability of the structure of the solutions.

Putting together equivalence relations and graph enumeration problems can give rise to new research directions. We worked on a particular flavor of AND/OR graphs due to their close relationship with dynamic programming problems. Outside of a DP setting, in the area of graph algorithms, the enumeration of subgraphs of a graph satisfying some particular properties, of which the enumeration of color classes of an ad-AND/OR graph is a special case, is also a challenging problem, and has stimulated interesting development – both theoretical and experimental – in Network biology [ASA19; Geo+09], Data mining [Mou+14], Data bases theory [Zen+06], etc. In these

applications, one typically wishes to understand the data, which is the graph itself, by exhibiting its patterns, represented as subgraphs satisfying some desired properties (e.g., connectivity, density). For the purpose of discovering distinctive patterns, an equivalence relation can be introduced, and the new question is then how to efficiently enumerate subgraphs that satisfy the desired properties and that are not equivalent to each other.

In the last part of Section 4.4.2, we hinted that our approach can be used in the context of graph optimization problems, together with the tree decomposition technique and DP-algorithms on tree decomposition. For a specific graph problem such as DOMINATING SET, one might want to define an equivalence relation that is more tailored to the problem and not based on tree decomposition. More generally, the most suitable equivalence relation on the set of solutions of a problem could be based on some non-local properties, and our framework would not apply. A recursive algorithm such as Algorithm 3 works well when the property of interest is local in some sense (here, in the form of an e-coloring). In order to efficiently enumerate equivalence classes that are based on non-local properties, we have to develop new techniques. Indeed, in Chapter 2, the enumeration of V-equivalence classes, which is based on the non-local property of event vector, was done using an entirely different method.

Chapter 5

Maximum agreement forest

5.1 Introduction

A natural question that arises in phylogenetic analysis is the quantification of dissimilarity between different (i.e., non isomorphic, see Chapter 1, Definition 1.2) phylogenetic trees on the same set of taxa. The difference in the tree topology can be a consequence of different inference methods, or different data sources being used (e.g., morphological data, behavioral data, genetic data, etc). Furthermore, regardless of the inference method or the data source, the constructed tree may not correctly represent evolutionary relationships, because not all groups of species follow a simple tree-like evolutionary pattern. The non tree-like evolutionary processes, such as hybridization, recombination, and horizontal gene-transfer, are collectively known as *reticulation events* [HRS10]. Due to reticulation events, phylogenetic trees representing the evolutionary history of different genes found in the same set of species may differ.

Several distance metrics are commonly used to quantify the difference between phylogenetic trees. The Robinson-Foulds distance is a popular metric that can be calculated in linear time [RF81; Day85]. The subtree prune-and-regraft (SPR) distance and the hybridization number [Bar+05] are more biologically meaningful, but are NP-hard to compute [AS01; BS05; BS07] (and even APX-hard [Hei+96; Rod03]). The SPR distance is particularly interesting as it provides a lower bound on the number of retic-

ulation events required to transform one tree to the other, which is a simple explanation for the difference between trees [BH06].

The MAXIMUM AGREEMENT FOREST PROBLEM (MAF PROBLEM) is closely related to the problem of computing the SPR distance [Hei+96]. In the case of two rooted directed binary trees, the optimal value of the MAF PROBLEM (using a subtle redefinition by [BS05]) coincides with the value of the rSPR distance, that is, the rooted variant of the SPR distance. In this chapter, we will focus on the MAF PROBLEM.

For the MAF PROBLEM, several standard approaches for dealing with NP-hard problems have been employed, including approximation algorithms, fixed-parameter algorithms [WZ09; WBZ13], and integer linear programming [Wu08]. Much of the literature has been devoted to the search of better approximation algorithms, leading to a succession of algorithms with ever-improving approximation ratios. The first correct approximation algorithm is a 5-approximation [Bon+06]. Subsequently, several 3-approximations have been proposed [BMS08; RSW07; WZ09], each one improving on the running time. The approximation ratio was later improved to 2.5 by [Shi+16] and then to 7/3 by [CMW16]. To date, the best approximation ratio of 2 is achieved by [SZS16] and [Che+20] which independently gave two algorithms using entirely different methods. Recently, [YCW19] made use of *Monte Carlo tree search*, improving on the 2-approximation of [Che+20] to achieve a better practical result.

In this chapter, we define formally the MAF PROBLEM and present a *heuristic* for it, that is, an algorithm that provides an approximate solution without a guaranteed approximation ratio, then we compare this algorithm experimentally with the existing approximation algorithms and show that it outperforms all but one of them in terms of the quality of the approximation. We conjectured that this algorithm also achieves the approximation ratio of 2. If this were true, we would have yet another 2-approximation, again based on entirely different ideas from the two existing ones, that is easier to understand and to implement and performs better in practice.

5.2 Definitions

Basic definitions about trees and forests can be found in Chapter 1, Section 1.2.2. In this chapter, the central object that we study is a X-forest, or a leaf-labeled rooted binary forest. It is a graph whose connected components are rooted binary trees, and its leaves are bijectively labeled by a set X. To be more consistent with the terminology commonly found in the literature (where the vocabulary associated with undirected trees and forests is most often used), in this chapter, the set of arcs of a forest F will be denoted by $E(F)$; nodes and arcs will be called respectively vertices and edges. Recall from Chapter 1 that if v is a vertex in a forest, $p(v)$ denotes its parent.

The edge incident on a leaf is called the *pendant edge* of that leaf. An *isolated vertex* in a rooted binary forest is a leaf that is also the root of a component (i.e., an out-degree zero and in-degree zero vertex). A *pendant subtree* of an X-forest is a subtree that can be obtained from the forest by deleting at most one edge. It is necessarily a rooted binary tree. Unless explicitly stated otherwise, we simply use the term *subtree* to designate a pendant subtree. A *cherry* is a subtree consisting of only three vertices: two leaves and their common parent. A cherry containing two leaves a and b is denoted by (a, b), or equivalently, (b, a).

Definition 5.1 (Leaf pair). Let (F_1, F_2) be a pair of X-forests, i.e., two rooted binary forests whose leaves are labeled by the same set X. A *leaf pair* (v_1, v_2) of (F_1, F_2) is an ordered pair consisting of a leaf v_1 of F_1 and a leaf v_2 of F_2 having the same label.

Definition 5.2 (Cutting an edge from an X-forest). Let F be an X-forest. Let $\mathcal{E} \subseteq E(F)$ be a set of edges of F. Then $\textsc{cut}(F, \mathcal{E})$ is an X-forest obtained from F by applying the following operations:

1. Remove from $E(F)$ the edges in $\mathcal{E}$.
2. Repeatedly contract each out-degree one and in-degree one vertex, i.e., remove it and then add an edge between its parent and its unique child.
3. Remove all out-degree one and in-degree zero vertices (their unique child then becomes the new root of that component).

When $\mathcal{E}$ contains only one edge e, $\textsc{Cut}(F, \mathcal{E})$ can be simply written as $\textsc{Cut}(F, e)$. The operation of cutting an edge is also called *making a cut*.

Definition 5.3 (AF, MAF). An *agreement forest* (AF) of two X-forests F_1 and F_2 is an X-forest F such that there exist $\mathcal{E}_1 \subseteq E(F_1)$ and $\mathcal{E}_2 \subseteq E(F_2)$ satisfying

$$F \cong \textsc{Cut}(F_1, \mathcal{E}_1) \cong \textsc{Cut}(F_2, \mathcal{E}_2)\,.$$

By choosing $\mathcal{E}_1 = E(F_1)$ and $\mathcal{E}_2 = E(F_2)$, we can always obtain an AF of F_1 and F_2 whose components are all isolated vertices.

A *maximum agreement forest* (MAF) is an AF whose order is minimum.

For a pair of X-trees (T_1, T_2) (i.e., two X-forests of order 1), $m(T_1, T_2)$ denotes the order of a MAF minus one, that is,

$$m(T_1, T_2) := \min\{|F| - 1 : F \text{ is an AF of } T_1 \text{ and } T_2\}\,. \tag{5.1}$$

Since each cut increases the number of components by one, the number $m(T_1, T_2)$ can also be interpreted as the number of edges that one needs to cut in each tree to produce an agreement forest.

The MAF PROBLEM can be stated as follows: given two X-trees T_1 and T_2, find the value of $m(T_1, T_2)$.

It is known that $m(T_1, T_2)$ is equal to the rSPR distance if the following preprocessing operations have been applied to each of the two input X-trees: (1) add a dummy leaf labeled by ρ (a symbol that is not already in the label set X), (2) add ρ to the label set X, and (3) add a dummy root whose two children are the dummy leaf ρ and the original root of the tree [BS05]. As our interest in solving the MAF PROBLEM originates from the computation of the rSPR distance, we will assume that this preprocessing step has already been applied. For convenience, we will refer to the optimal value of an instance of the MAF PROBLEM simply as the *distance*. Therefore, the value of $m(T_1, T_2)$ defined in Equation (5.1) is called the distance between T_1 and T_2.

Even though an instance of the MAF PROBLEM is a pair of trees, for the purpose of analyzing our algorithm, it is worthwhile to extend the notion of distance to a pair of forests.

Definition 5.4 (Distance between a pair of X-forests). Let (F_1, F_2) be a pair of X-forests. Let (d_1, d_2) be an ordered pair of integers satisfying $d_2 = d_1 + |F_1| - |F_2|$.

- We say that (d_1, d_2) is an *upper bound of the distance*, denoted by $m(F_1, F_2) \leq (d_1, d_2)$, if there exists an AF F of F_1 and F_2 such that $|F| \leq d_1 + |F_1| = d_2 + |F_2|$. Equivalently, $m(F_1, F_2) \leq (d_1, d_2)$ if there exists $\mathcal{E}_1 \subseteq E(F_1)$ of size at most d_1 and $\mathcal{E}_2 \subseteq E(F_2)$ of size at most d_2 such that $\textsc{Cut}(F_1, \mathcal{E}_1) \cong \textsc{Cut}(F_2, \mathcal{E}_2)$.
- We say that (d_1, d_2) is a *lower bound of the distance*, denoted by $m(F_1, F_2) \geq (d_1, d_2)$, if for all AF F of F_1 and F_2, the order of F satisfies $|F| \geq d_1 + |F_1| = d_2 + |F_2|$.
- If (d_1, d_2) is an upper bound and a lower bound of the distance, we simply call it *the distance* between F_1 and F_2, and write $m(F_1, F_2) = (d_1, d_2)$.

It is clear that if $|F_1| = |F_2| = 1$, the distance (d_1, d_2) satisfies $d_1 = d_2 = |F| - 1$ for any MAF F of F_1 of F_2. The definition of $m(F_1, F_2)$ coincides with the one given in Equation (5.1) for a pair of trees, but with an important subtlety: the distance between a pair of forests is a pair of integers, while the distance between a pair of trees is a single integer. As we will see in the next section, the algorithm needs to work on pairs of forests with "unbalanced" numbers of components, i.e., (F_1, F_2) with $|F_1| \neq |F_2|$, hence, different numbers of cuts are needed in each forest before reaching a MAF.

The following easy lemmas are useful for expressing the quality of a cut in terms of the distance. As the distance can be intuitively understood as the number of cuts that still need to be made before reaching a MAF, a cut in one forest decreases (the corresponding element of) the distance by either zero or one; a cut is effective if it indeed decreases the distance by one.

Lemma 5.1 (Making cuts does not increase the distance). *Let (F_1, F_2) be a pair of X-forests. Let $\mathcal{E}_1 \subseteq E(F_1)$ and $\mathcal{E}_2 \subseteq E(F_2)$ be some subsets of edges of F_1 and F_2. Then*

$$m(\textsc{Cut}(F_1, \mathcal{E}_1), \textsc{Cut}(F_2, \mathcal{E}_2)) \geq (d_1 - |\mathcal{E}_1|, d_2 - |\mathcal{E}_2|)$$

where $(d_1, d_2) = m(F_1, F_2)$.

Proof. Any AF F of $\textsc{Cut}(F_1, \mathcal{E}_1)$ and $\textsc{Cut}(F_2, \mathcal{E}_2)$ is also an AF of F_1 and F_2. So $|F| \geq d_1 + |F_1| = d_1 - |\mathcal{E}_1| + |\textsc{Cut}(F_1, \mathcal{E}_1)|$ and $|F| \geq d_2 + |F_2| = d_2 - |\mathcal{E}_2| + |\textsc{Cut}(F_2, \mathcal{E}_2)|$. □

Lemma 5.2 (A cut is effective if it is used by a MAF). *Let* (F_1, F_2) *be a pair of* X*-forests with* $m(F_1, F_2) = (d_1, d_2)$.

- *(Cutting in only one forest) Let* e_1 *be an edge of* F_1*. If there exists a MAF* F *of* F_1 *and* F_2 *such that* $e_1 \in \mathcal{E}_1$*, where* $\mathcal{E}_1$ *is the subset of edges of* F_1 *such that* $F \cong \mathit{CUT}(F_1, \mathcal{E}_1)$*, then*

$$m(\mathit{CUT}(F_1, e_1), F_2) = (d_1 - 1, d_2).$$

- *(Cutting in both forests) Let* e_1 *be an edge of* F_1 *and* e_2 *be an edge of* F_2*. If there exists a MAF* F *of* F_1 *and* F_2 *such that* $e_1 \in \mathcal{E}_1$ *and* $e_2 \in \mathcal{E}_2$*, where* $\mathcal{E}_1$ *and* $\mathcal{E}_2$ *are the subsets of edges of* F_1 *and* F_2 *such that* $F \cong \mathit{CUT}(F_1, \mathcal{E}_1) \cong \mathit{CUT}(F_2, \mathcal{E}_2)$*, then*

$$m(\mathit{CUT}(F_1, e_1), \mathit{CUT}(F_2, e_2)) = (d_1 - 1, d_2 - 1).$$

Proof. (For the first statement only; the proof of the second statement is similar) By Lemma 5.1, $m(\mathrm{CUT}(F_1, e_1), F_2) \geq (d_1 - 1, d_2)$. For the other direction, since $e_1 \in \mathcal{E}_1$, the forest F is an AF of $\mathrm{CUT}(F_1, e_1)$ and F_2. We have $|F| = d_1 + |F_1| = (d_1 - 1) + |\mathrm{CUT}(F_1, e_1)|$, so $m(\mathrm{CUT}(F_1, e_1), F_2) \leq (d_1 - 1, d_2)$. □

5.3 The algorithm

5.3.1 Reduction rules

Given a pair of X-trees T_1 and T_2, we seek an approximate solution of their distance, that is, an upper bound of $m(T_1, T_2)$. The general strategy of the algorithm is straightforward. At any step, we have a pair of X-forests and we cut edges in both of them until the pair becomes isomorphic; the result is an AF and the final number of cuts is an upper bound of the distance. Notice that not all algorithms in the literature follow this approach, for example, the 3-approximation of [WZ09] only cuts edges in one of the two input trees.

The first important component of the algorithm is the reduction rules. These rules tell us how to transform a pair of forests into another one with strictly fewer leaves while *preserving the distance*. Intuitively, these rules correspond to the observations of the type "it is forbidden (or mandatory) to cut such an edge".

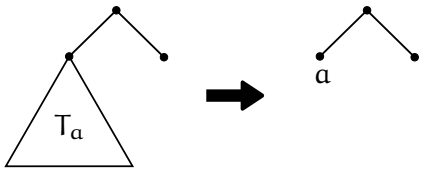


Figure 5.1: Illustration of the reduction rule 1. Only the relevant and common structure of the two forests is drawn.

Four reduction rules are considered in the algorithm. The first two were defined in [AS01; BS05]. We will only give the proof of correctness for the other two. The first three rules are concerned with common structures that can be found in both forests: those structures can be replaced by simpler ones. The fourth rule is of a different nature and identifies an edge that must be cut in exactly one of the two forests.

- **Reduction rule 1** (Subtree reduction): Replace any subtree of more than one leaf that occurs in both forests by a single leaf with a new label.
- **Reduction rule 2** (Chain reduction): Replace any chain of subtrees that occurs identically and with the same relative orientation in both forests by three new leaves with new labels correctly oriented to preserve the direction of the chain.
- **Reduction rule 3** (Root-chain reduction): Replace any chain of subtrees that occurs identically and with the same relative orientation in both forests and starts from the root of its component by a single leaf with a new label.
- **Reduction rule 4** (Isolated vertex reduction): If there exists a leaf pair (v_1, v_2) such that v_1 (resp. v_2) is an isolated vertex but v_2 (resp. v_1) has a parent, cut the pendant edge of v_2 (resp. v_1).

Following what the authors did in [AS01; BS05], we do not formally define what is a *chain* in the rule 2 but only show it graphically. Illustrations of application of each of the first three reduction rules are shown in Figures 5.1–5.3.

It is worth mentioning that we will always call the forests X-forests in any step of the algorithm, although the original label set X may have changed as a consequence of applying the reduction rules.

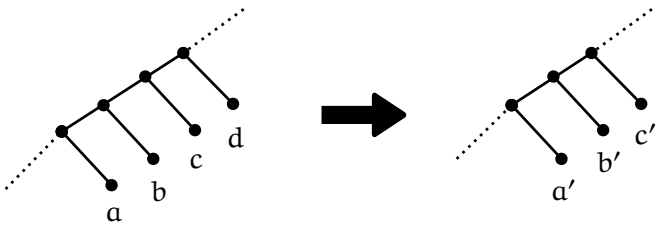


Figure 5.2: Illustration of the reduction rule 2. Only the relevant and common structure of the two forests is drawn. Common subtrees have already been replaced by leaves after applying the reduction rule 1.

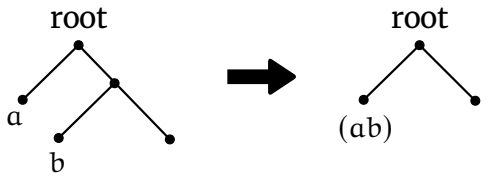


Figure 5.3: Illustration of the reduction rule 3. Only the relevant and common structure of the two forests is drawn. Common subtrees have already been replaced by leaves after applying the reduction rule 1.

Lemma 5.3 (The reduction rule 3 preserves the distance). *Let* (F_1, F_2) *be a pair of* X*-forests. If the reduction rule 3 can be applied and the result is a pair* (F_1', F_2')*, then* $m(F_1, F_2) = m(F_1', F_2')$.

Proof. It suffices to consider two leaf pairs (a_1, a_2) and (b_1, b_2) (labeled respectively by a and b) such that $p(b_1) = p(p(a_1))$ and $p(b_1)$ is the root of a component of F_1, and $p(b_2) = p(p(a_2))$ and $p(b_2)$ is the root of a component of F_2 (see Figure 5.3). Let (ab) be the label of the new leaf in (F_1', F_2').

First, let F' be a MAF of F_1' and F_2'. If the leaf (ab) is isolated in F', replace it by a cherry of two leaves a and b; otherwise, the parent of (ab) is the root of a component, and we can replace (ab) by a leaf b, and add a new root and connect it to the old root $p(a)$ and to a leaf a. The result is an AF of F_1 of F_2, hence, $m(F_1, F_2) \leq m(F_1', F_2')$.

For the other direction, let F be a MAF of F_1 and F_2. We first show that the leaves labeled by a and b are in the same component of F. Assume that a and b are in different components. Then a must be an isolated vertex; b is either an isolated vertex or $p(b)$ is the root of a component. In the first case, connect the two isolated vertices a and b to form a cherry. In the second case, add a new root to the component containing b,

connecting it to the old root $p(b)$ and to the isolated vertex a. In both cases, we get an AF of F_1 of F_2 of order strictly smaller than F, contradicting the minimality of the order of F. Therefore, a and b are in the same component in F. Replace the chain containing a and b by a new leaf labeled by (ab). The result is an AF of F'_1 and F'_2, hence, $m(F'_1, F'_2) \leq m(F_1, F_2)$. □

Lemma 5.4 (The cut made by the reduction rule 4 is effective). *Let* (F_1, F_2) *be a pair of* X*-forests. If the reduction rule 4 cuts an edge* e_1 *in* F_1 *(resp.* e_2 *in* F_2*), then*

$$m(\mathit{CUT}(F_1, e_1), F_2)) = (d_1 - 1, d_2)$$
$$(\textit{resp. } m(F_1, \mathit{CUT}(F_2, e_2))) = (d_1, d_2 - 1))$$

where $(d_1, d_2) = m(F_1, F_2)$.

Proof. The proof is trivial: if a leaf appears as an isolated vertex in either F_1 or F_2, it must appear as an isolated vertex in every MAF of F_1 and F_2. □

During the algorithm, the reduction rules will be applied at the very beginning as well as at the end of each iteration. The remaining steps of an iteration thus only consider a pair of forests that is irreducible.

Definition 5.5 (Irreducible pair of forests). A pair of X-forests is called *irreducible* if none of the reduction rules 1–4 can be applied.

The next lemma says that a pair of forests can be made irreducible efficiently, and the size of an irreducible pair of forests is bounded linearly by their distance. The proof is omitted and follows from the similar statements in [AS01; BS05] (the addition of the two new reduction rules does not affect the analysis).

Lemma 5.5. *A pair of* X*-forests* (F_1, F_2) *of size* n *(i.e., the number of vertices in each forest) can be transformed into an irreducible pair* (F'_1, F'_2) *of size* n' *in* $O(n)$ *time, and the size* n' *is bounded linearly by* $m(F_1, F_2)$.

5.3.2 Common continuous subtree (CCS)

Definition

The central idea of the algorithm is to identify subtrees that are "almost" identical in the pair of forests. This is in a similar spirit as the first reduction rule, which identifies subtrees that are exactly identical. In our definition of "almost identical subtrees", we allow at most one cut in either forest, that is to say, after cutting at most one edge, that subtree will occur in both forests and the reduction rule 1 applies. The next few definitions express this formally.

Definition 5.6 (CS). Let F be a X-forest. A *continuous subtree* (CS) of F is a subtree $\mathcal{H}$ of F having at least two leaves, and in which at most one edge (u, v) is marked as the *cut edge* and satisfies $u \neq r(\mathcal{H})$ (i.e., the cut edge is not incident to the root of the CS).

When no edge is marked as the cut edge, the cut edge of the CS is defined to be the empty set $\emptyset$. Accordingly, we define $\textsc{Cut}(F, \emptyset) := F$.

Definition 5.7 (CCS). Let (F_1, F_2) be a pair of X-forests. A *common continuous subtree* (CCS) of (F_1, F_2) is an ordered pair $(\mathcal{H}_1, \mathcal{H}_2)$ such that

- $\mathcal{H}_1$ is a CS of F_1, $\mathcal{H}_2$ is a CS of F_2.
- Let e_1 and e_2 be the cut edges of $(\mathcal{H}_1, \mathcal{H}_2)$. Then $\textsc{Cut}(\mathcal{H}_1, e_1) \cong \textsc{Cut}(\mathcal{H}_2, e_2)$.

The number of leaves in $\textsc{Cut}(\mathcal{H}_1, e_1)$ or equivalently in $\textsc{Cut}(\mathcal{H}_2, e_2)$ is called the *size* of the CCS.

From the definition of a CS, after performing the cut, it must still remain at least two leaves in the CS, hence, the size of a CCS is at least two. Consequently, $\textsc{Cut}(\mathcal{H}_1, e_1)$ and $\textsc{Cut}(\mathcal{H}_2, e_2)$ share at least one common cherry.

Classification

In an **irreducible** pair of forests, the possible sizes or topologies of a CCS are very limited. As we just pointed out, after performing the cuts in a CCS, a common cherry appears in both forests. This common cherry must not exist before the cuts, otherwise

the first reduction rule would apply. It implies that, in at least one of the two forests, the cut edge of the CCS is the edge that "separates" the two leaves in that cherry. The next lemma provides a list of properties of CCSes in an irreducible pair of forests, and classifies all possible CCSes into three *types* in relation to the cut edge that "separates the new common cherry".

Lemma 5.6. *Let* $(\mathcal{H}_1, \mathcal{H}_2)$ *be a CCS of a pair of irreducible* X*-forests* (F_1, F_2)*. Then*

1. *Th cut edges of* $\mathcal{H}_1$ *and* $\mathcal{H}_2$ *cannot both be empty.*

2. *For either* $\mathcal{H}_1$ *or* $\mathcal{H}_2$ *or for both, the cut edge* (u, v) *is incident to the pendant edge of a leaf* a*, that is,* $u = p(a)$*.*

   A CS satisfying this property is called a leading *CS of that CCS.*

3. *Assume wlog that* $\mathcal{H}_1$ *is a leading CS of the CCS. Let* a *be leaf adjacent to the cut edge of* $\mathcal{H}_1$*, and let* b *be the leaf such that* $p(b) = p(p(a))$ *(such a leaf* b *exists because there must be a cherry* (a, b) *after cutting the cut edge of* $\mathcal{H}_1$*). Let* a' *and* b' *be the leaves of* F_2 *having the same labels as* a *and* b*. Then exactly one of following is true:*

   (a) (a', b') *is a cherry of* $\mathcal{H}_2$*. In this case,* $(\mathcal{H}_1, \mathcal{H}_2)$ *is called a CCS* of type one.

   (b) $\mathcal{H}_2$ *has a cut edge* (u, v) *such that* $u = p(b')$ *and* $p(u) = p(a')$*. In this case,* $(\mathcal{H}_1, \mathcal{H}_2)$ *is called a CCS* of type two.

   (c) $\mathcal{H}_2$ *has a cut edge* (u, v) *such that* $u = p(a')$ *and* $p(u) = p(b')$*. In this case,* $(\mathcal{H}_1, \mathcal{H}_2)$ *is called a CCS* of type three.

   In any of the three cases, the leaves a *and* b *in* F_1 *and the leaves* a' *and* b' *in* F_2 *are said to be forming the* target cherry *of the CCS.*

4. *The maximum size of a CCS of type one is 6. The maximum size of a CCS of type two is 5. The maximum size of a CCS of type three is 3.*

Proof. 1. Suppose that both cut edges are empty, then $\mathcal{H}_1$ and $\mathcal{H}_2$ are common subtrees of F_1 and F_2. Since the reduction rule 1 does not apply, any common subtree can only have one leaf. By definition, a CS has at least two leaves.

2. Suppose that neither of the two CSes is a leading CS, then there exists a common cherry in $\textsc{Cut}(\mathcal{H}_1, e_1)$ and $\textsc{Cut}(\mathcal{H}_2, e_2)$ that is unchanged by performing the cuts. This cherry is a common subtree of F_1 and F_2 and contradicts the irreducibility.

3. Trivial from the definition of a CCS.

4. The limits on the size of a CCS come from the fact that neither reduction rule 1 nor reduction rule 2 applies. The assertion can be checked by listing exhaustively all possible configurations of a CCS. In Figure 5.4, we provided such a list. The figure should be read as follows. The two leaves forming the target cherry are labeled by a and b. The row indexed by 0 shows the possible configurations of the leading CS of a CCS. Once a leading CS has been chosen in row 0, in the same column, the rows 1–3 provide possible configurations of the other CS (leading or not), such that these two CSes together form a correct CCS. Each column, indexed from 2 to 6, shows the possible configurations for a given size of a CCS (starting from column 3, the row 2 splits into two sub-rows). A possible configuration of a CCS is a **combination** of one configuration from row 0 with one configuration from rows 1–3, both taken from the same column. The choice between rows 1–3 determines whether the CCS is of type one, two, or three.

□

The classification of CCSes allows us to find CCSes very efficiently in irreducible pairs of forests. As a key operation in the algorithm, CCS searching is needed repeatedly and crucially impacts the running time. While our definition of CCS is simple and interesting by itself, the assumption of irreducibility makes it especially interesting from a practical point of view.

Lemma 5.7. *Given an irreducible pair of forests, the list of its CCSes can be found in linear time in the size of the forests.*

Proof. It suffices to identify all target cherries (a, b) which appear either as cherries or as "cherries separated by an edge". There can only be a linear number of them, hence, all CCSes of size 2 can be found in linear time. Any CCS of size larger than 2 can be

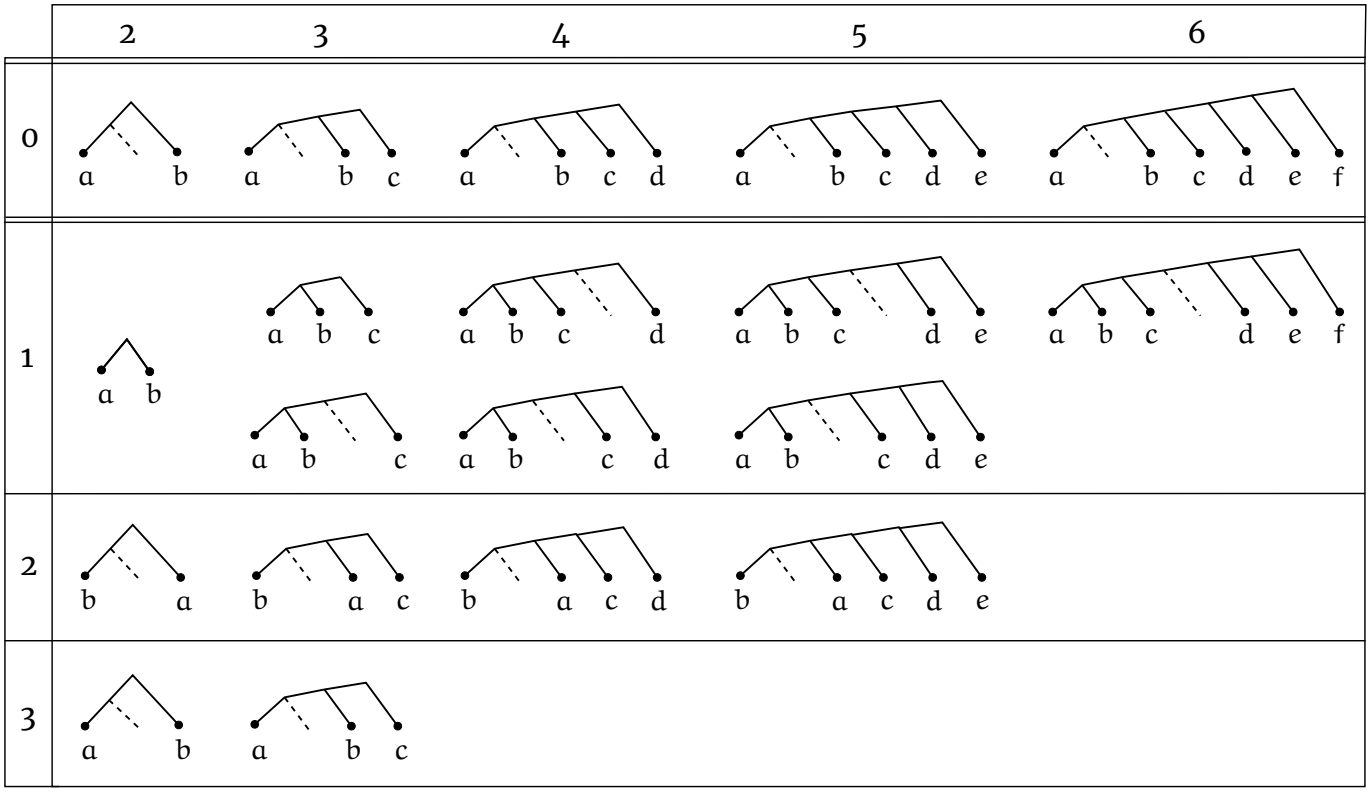


Figure 5.4: Classification of CCSes. A possible configuration of a CCS is a **combination** of one configuration from the row indexed by 0, with one configuration from rows 1–3, both taken from the same column. The choice between rows 1–3 determines whether the CCS is of type one, two, or three. The column index indicates the size of the CCS.

obtained from a CCS of size 2 by successively adding leaves that are next to the target cherry (see Figure 5.4); this clearly requires constant time. □

Best CCS and best pre-CCS leaf pair

At each iteration, the algorithm identifies all CCSes of the irreducible pair of forests. Two situations can then occur. If there are more than one CCS, it selects one of them. If, on the other hand, there exists no CCS, it in turn tries to find a pre-CCS leaf pair, which we will define later; as the name suggests, after cutting out such a leaf pair from the forests, some CCSes will emerge as a result. In both situations, the choice of the CCS or the pre-CCS leaf pair is guided by a partial order between CCSes, taking into account the type and the size.

Definition 5.8 (Best CCS). For a given set of CCSes, consider the following partial order

(reflexive, antisymmetric, and transitive) "better than":

- A CCS of type one is better than a CCS of type two.
- A CCS of type two is better than a CCS of type three.
- For two CCSes of the same type, a CCS of larger size is better than the smaller one.

A CCS is called a *best CCS* if it is a minimal element, i.e., there exists no other better CCS in the set.

When the pair of forests does not have any CCS, the algorithm chooses to cut a leaf pair, or more precisely, the pendant edges of a leaf pair. Indeed, in the special case of an irreducible pair of forests without any CCS, this kind of cut preserves the irreducibility. This allows the algorithm to "look one step ahead" and to identify new CCSes immediately after the cut, i.e., without applying the reduction rules and going to the next iteration.

Lemma 5.8. *Let* (F_1, F_2) *be a pair of irreducible* X*-forests that does not have any CCS. Let* (v_1, v_2) *be a leaf pair of* (F_1, F_2)*. Then, after cutting the pendant edges* e_1 *and* e_2 *of* v_1 *and* v_2*, the pair of forests* $(\mathrm{CUT}(F_1, e_1), \mathrm{CUT}(F_2, e_2))$ *are still irreducible.*

Proof. It is easy to check that the reduction rules 2 and 3 cannot apply to the pair $(\mathrm{CUT}(F_1, e_1), \mathrm{CUT}(F_2, e_2))$. To see that the Reduction rule 1 does not apply, suppose that $(\mathrm{CUT}(F_1, e_1), \mathrm{CUT}(F_2, e_2))$ has one common subtree of at least two leaves. Let a and b be two leaves that form a common cherry of $\mathrm{CUT}(F_1, e_1)$ and $\mathrm{CUT}(F_2, e_2)$. The leaves a and b cannot appear as a cherry in both forests F_1 and F_2 because the pair (F_1, F_2) is irreducible. Thus at least one edge e_1 or e_2 must separate the two leaves. Assume wlog that e_1 is incident to the pendant edge of a in F_1. Let $\mathcal{H}_1$ be the CS of F_1 rooted at the parent of b with e_1 marked as the cut edge. Depending on the position of the edge e_2 relative to the cherry (a', b') in F_2 (three cases: outside of the cherry, incident to the pendant edge of b', or incident to the pendant edge of a'), we can choose the CS $\mathcal{H}_2$ accordingly (each case corresponds to a type of CCS) such that $(\mathcal{H}_1, \mathcal{H}_2)$ is a CCS of (F_1, F_2) (a and b form the target cherry of that CCS), contradicting the assumption that the pair (F_1, F_2) does have any CCS. □

Definition 5.9 (Pre-CCS leaf pair). Let (F_1, F_2) be a pair of irreducible X-forests that does not have any CCS. A leaf pair (v_1, v_2) of (F_1, F_2) is called a *pre-CCS leaf pair* if, after cutting the pendant edges e_1 and e_2 of v_1 and v_2, the new pair of forests $(\text{CUT}(F_1, e_1), \text{CUT}(F_2, e_2))$ has at least one CCS.

Definition 5.10 (Best pre-CCS leaf pair). Let (F_1, F_2) be a pair of irreducible X-forests that does not have any CCS. Consider the set of CCSes

$$\widetilde{\mathcal{H}} := \bigcup_{(e_1,e_2)} \left\{ (\mathcal{H}_1, \mathcal{H}_2) : (\mathcal{H}_1, \mathcal{H}_2) \text{ is a best CCS of } (\text{CUT}(F_1, e_1), \text{CUT}(F_2, e_2)) \right\}$$

where e_1 and e_2 are the pendant edges of v_1 and v_2, and the union is taken over all pre-CCS leaf pairs (v_1, v_2) of (F_1, F_2). A leaf pair (v_1, v_2) for which a best CCS $(\mathcal{H}_1, \mathcal{H}_2)$ of $(\text{CUT}(F_1, e_1), \text{CUT}(F_2, e_2))$ is a minimal element of $\widetilde{\mathcal{H}}$ for the relation "better than" (see Definition 5.8) is called a *best pre-CCS leaf pair* associated with a best CCS $(\mathcal{H}_1, \mathcal{H}_2)$.

From this definition, if there exist two distinct best pre-CCS leaf pairs associated respectively with the best CCSes $(\mathcal{H}_1, \mathcal{H}_2)$ and $(\mathcal{H}'_1, \mathcal{H}'_2)$, then both best CCSes are of the same type and of the same size, although they are CCSes in two different pairs of forests.

5.3.3 Summary of the algorithm

The algorithm, shown in Algorithm 5, is called NewCCS. The name acknowledges the fact that it originated from ideas of algorithms that were previously explored in the team and in which the term CCS, though defined differently, was first introduced.

At each iteration, the algorithm updates a pair of forests (F_1, F_2) and an integer k that keeps track of the number of cuts made so far in F_1. The edges that are cut from F_1 by the algorithm come from three sources: either it is the result of applying the reduction rule 3 (Line 21), or it is the pendant edge of a leaf (Lines 13 and 19), or it is the cut edge of a CCS (Lines 9 and 15). As the cut edge e of a CCS may not exist (defined to be $\emptyset$ in this case), we use the notation $|e|$ that equals zero if the edge is empty and equals one otherwise. At least one edge is cut in one of the forests in each iteration (by Lemma 5.6, the two cut edges of a CCS cannot both be empty), so the **while** loop necessarily stops and ends with a pair of isomorphic forests. As the reduction rules

Algorithm 5: (NewCCS) Approximate $m(T_1, T_2)$

Input: *A pair of* X*-trees* T_1 *and* T_2
Output: *An integer* k *such that* $m(T_1, T_2) \leq k$
$F_1 \leftarrow T_1$; $F_2 \leftarrow T_2$; $k \leftarrow 0$
Make (F_1, F_2) irreducible by applying the reductions rules
while F_1 *and* F_2 *are not isomorphic* **do**
    if *there exists a CCS of* (F_1, F_2) **then**
        Let $(\mathcal{H}_1, \mathcal{H}_2)$ be a best CCS of (F_1, F_2)
        Let e_1 and e_2 be the cut edges of $(\mathcal{H}_1, \mathcal{H}_2)$
        $F_1 \leftarrow \text{CUT}(F_1, e_1)$; $F_2 \leftarrow \text{CUT}(F_2, e_2)$; $k \leftarrow k + |e_1|$
    else if *there exists a pre-CCS leaf pair of* (F_1, F_2) **then**
        Let (v_1, v_2) be a best pre-CCS leaf pair of (F_1, F_2) associated with a best CCS $(\mathcal{H}_1, \mathcal{H}_2)$
        Let e_1 and e_2 be the pendant edges of v_1 and v_2
        $F_1 \leftarrow \text{CUT}(F_1, e_1)$; $F_2 \leftarrow \text{CUT}(F_2, e_2)$; $k \leftarrow k + 1$
        Let e_3 and e_4 be the cut edges of $(\mathcal{H}_1, \mathcal{H}_2)$
        $F_1 \leftarrow \text{CUT}(F_1, e_3)$; $F_2 \leftarrow \text{CUT}(F_2, e_4)$; $k \leftarrow k + |e_3|$
    else // i.e., there is no CCS and no pre-CCS leaf pair
        Let (v_1, v_2) be a random leaf pair of (F_1, F_2)
        Let e_1 and e_2 be the pendant edges of v_1 and v_2
        $F_1 \leftarrow \text{CUT}(F_1, e_1)$; $F_2 \leftarrow \text{CUT}(F_2, e_2)$; $k \leftarrow k + 1$
    end
    Make (F_1, F_2) irreducible by applying the reductions rules
    Increment k with the number of edges the reduction rule 3 has cut in F_1
end
return k

preserve the distance, exactly k cuts are needed in either forests to arrive at an AF, therefore, k is an upper bound of the distance. The number of iterations is $O(n)$ and each iteration requires $O(n)$ time (to see this, notice that the number of leaf pairs is

$O(n)$ and cutting a leaf pair only creates $O(1)$ new CCSes), hence the next lemma.

Lemma 5.9. *Algorithm 5 returns an upper bound of* $m(T_1, T_2)$ *in* $O(n^2)$ *time.*

We end this section with the following conjecture:

Conjecture 5.1. *Algorithm 5 is a 2-approximation for computing* $m(T_1, T_2)$.

5.4 Experimental results

Algorithm 5 has been implemented in Python. Because there is some randomness, the implementation performs 10 random runs and outputs the best solution. In this section, the name NewCCS only refers to the best-out-of-ten implementation.

To compare with our NewCCS algorithm, we will use five existing algorithms that have a publicly available implementation.

- WZ (`https://github.com/cwhidden/rspr`), the 3-approximation by [WZ09] which runs in linear time.
- SVN (`https://nolver.net/maf/`), the 2-approximation by [SZS16]. It runs in quadratic time.
- CMW, CHN, CombMCTS (`http://rnc.r.dendai.ac.jp/rsprHN.html`), respectively, the (7/3)-approximation by [CMW16] (quadratic time), the 2-approximation by [Che+20] (cubic time), the Monte Carlo tree search algorithm by [YCW19], which is a practical improvement of the previous two (no complexity analysis).

5.4.1 Practical issues

Ideally, we would like to run all the above algorithms on all the available datasets. This was however not possible due to various runtime problems that we encountered when running the software.

- CMW can output a number that is smaller than the real distance. For example, for the following instance with distance 4, the output is 3.

```
(((((((((L1,L2),L3),L4),L5),L6),L7),L8),L9),L10);
(((((((((L1,L2),L4),L8),L9),L10),L5),L6),L3),L7);
```

- CMW can output a number that is more than 7/3 times the real distance, while it is a (7/3)-approximation. For example, for the following instance with distance 2, the output is 5.

```
(((((((((L1,L2),L3),L4),L5),L6),L7),L8),L9),L10);
(((((((((L2,L4),L5),L1),L6),L7),L3),L8),L9),L10);
```

- CHN can output a number that is smaller than the real distance. For example, for the following instance with distance 3, the output is 2 (however, the output is 3 if the order of two input trees is switched).

```
(((L5,L4),L3),(((L9,L7),((L10,L8),(L6,L2))),L1));
((((L8,((((L9,L7),(L10,L2)),L1),L5)),L4),L6),L3);
```

Any of the above problems will be referred to as an *error*. It should be clear that the possibility of running into errors limits the experiments and the analyses that we are able to perform.

We will now proceed to the comparison of NewCCS with the five existing algorithms. The datasets that we use are either from the literature or from our own construction. For any instance in the datasets, the exact distance is known (either by construction, or pre-computed using one of the available software, e.g., [WBZ13]). The experiments are run on a laptop PC with Intel i5-3380M CPU (2.90 GHz, 4 cores) and 8 GB RAM.

5.4.2 Comparison using randomly generated datasets

For the first experiment, we use the three datasets R_{100}^{50}, R_{200}^{80}, and R_{200}^{100} from [YCW19], each containing 120 instances. In R_a^b, each instance (T_1, T_2) is obtained by first generating a random phylogenetic tree T_1 with a leaves and then obtaining T_2 by applying b random rSPR operations. For example, an instance in R_{100}^{50} has size 100, and the distance is at most 50.

On these three datasets, all six algorithms run fast enough, and without encountering any error, so we can compare their approximation ratio as the authors did in [YCW19]. For each dataset, we computed the average and the maximum approximation ratios among the 120 instances. The results are shown in Table 5.1.

	Average Ratio			**Max Ratio**		
	R^{50}_{100}	R^{80}_{200}	R^{100}_{200}	R^{50}_{100}	R^{80}_{200}	R^{100}_{200}
WZ	1.431	1.478	1.436	1.692	1.671	1.587
SVN	1.494	1.560	1.501	1.705	1.725	1.658
CMW	1.148	1.088	1.141	1.419	1.286	1.317
CHN	1.136	1.084	1.127	1.356	1.225	1.276
CombMCTS	1.006	1.004	1.010	1.047	1.029	1.057
NewCCS	1.058	1.058	1.058	1.143	1.135	1.126

Table 5.1: For each of the six algorithms and for each dataset, the average and maximum approximation ratio on the 120 instances of the randomly generated datasets.

On the randomly generated datasets, the 3-approximation WZ produces slightly better ratios than the 2-approximation SVN. Those two algorithms are significantly outperformed by the (7/3)-approximation CMW and the 2-approximation CHN. The lowest average approximation ratio is achieved by CombMCTS, which was designed to improve the approximation ratio in practice by means of a Monte Carlo tree search technique. The NewCCS algorithm yields the second best average approximation ratio on all three datasets.

Instead of looking at the quality of the output of each algorithm on average on the whole dataset, we can ask, for each individual instance, how well is the output of one algorithm compared to the output of another. For this, we can compute the average difference of the approximation ratio, that is, the difference between the two output values divided by the real distance, averaged over the 120 instances. Take the dataset R^{100}_{200}. On average, NewCCS achieves an approximation ratio that is 0.083 better than

CMW, 0.069 better than CHN, and 0.048 worse than CombMCTS.

5.4.3 Comparison using a dataset of caterpillars of fixed size

We have constructed a dataset consisting of 1814399 pairs of trees with distances between 1 and 7, which correspond to all possible pairs of caterpillars with 10 leaves (a rooted caterpillar is a rooted binary tree with only one cherry). Since the number of instances in this dataset is quite large, we randomly selected a subset of 5000 instances (with distances also spanning from 1 to 7). On this subset, we ran all six algorithms and measured the running time. We also computed the average and the maximum approximation ratios. If an error occurs (see Section 5.4.1), the instance is excluded from the computation of the average and the max approximation ratios (for that algorithm only). The results are shown in Table 5.2.

	Time	Average Ratio	Max Ratio	Remarks
WZ	37 s	1.333	2.333	
SVN	9 s	1.430	2.0	
CMW	9 min 10 s	1.140	2.0	643 errors (1 ratio $> 7/3$, 642 ratio < 1)
CHN	11 min 01 s	1.183	2.0	9 errors (ratio < 1)
CombMCTS	23 min 32 s	1.020	1.5	
NewCCS	14 s	1.077	1.75	

Table 5.2: For each of the six algorithms, the running time, the average and maximum approximation ratio on the 5000 pairs of caterpillars with 10 leaves.

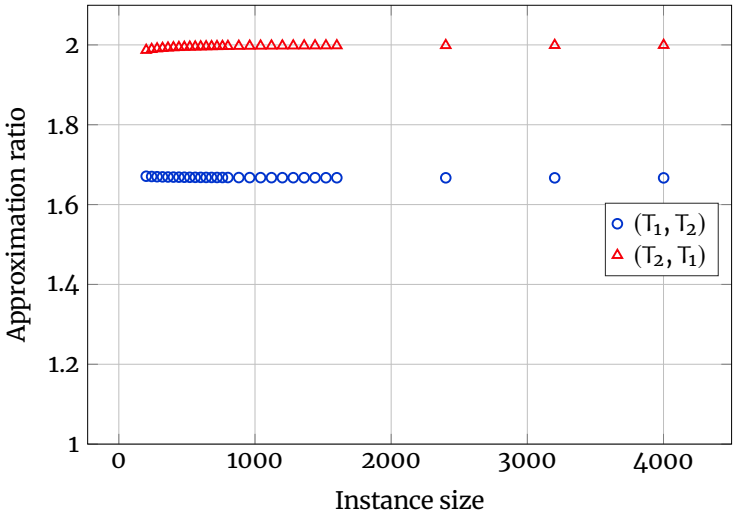
On this caterpillar dataset, both CMW and CHN run into some type of error. Out of the four remaining algorithms, SNV and WZ produce poorer average approximation ratios; CombMCTS gives the best approximation but is significantly slower; NewCCS is the second fastest and yields the second best average ratio. If we run the algorithms on the full dataset of 1814399 instances, the estimated running time of CombMCTS is 5.9 days, while NewCCS only needs 1.5 hour. Hence, for this dataset consisting of a large

number of instances of a very small size, NewCCS is of strong practical interest, as it offers a good compromise between the speed and the quality of the approximation.

5.4.4 Comparison using a special family of instances

For the last experiment, we used a dataset that we call HUGE. It consists of 58 pairs of trees of sizes between 201 and 4001. These instances belong to the $\mathcal{J}'$ family introduced by [RSW07] and for which the distances are known exactly. Precisely, for an instance of size $40q + 1$ in this dataset (where q is an integer), the distance is $15q + 1$. For every instance (T_1, T_2) in this dataset, another instance (T_2, T_1) is also included, that is, the same pair of trees with the order switched.

We ran all six algorithms on HUGE and recorded the approximation ratio and the running time. The results are plotted in Figures 5.5–5.10. The X-coordinates correspond to the sizes of the instances. For each size, there are two points (blue circle and the red triangle) which correspond to the two instances that only differ by the ordering of the two input trees.



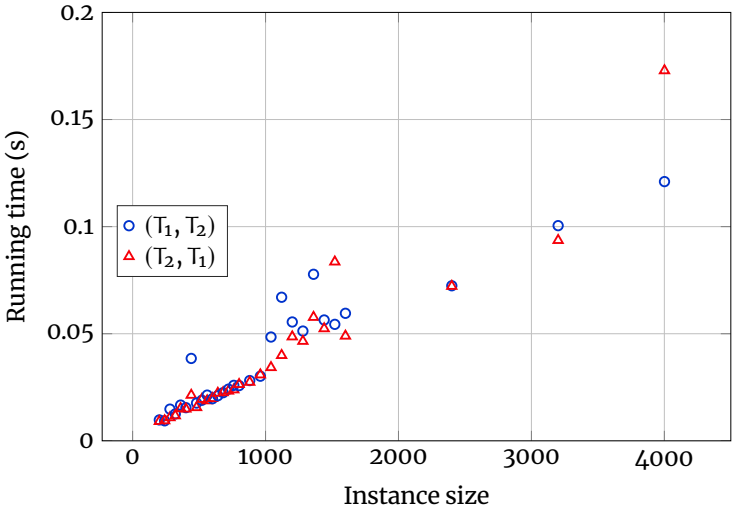
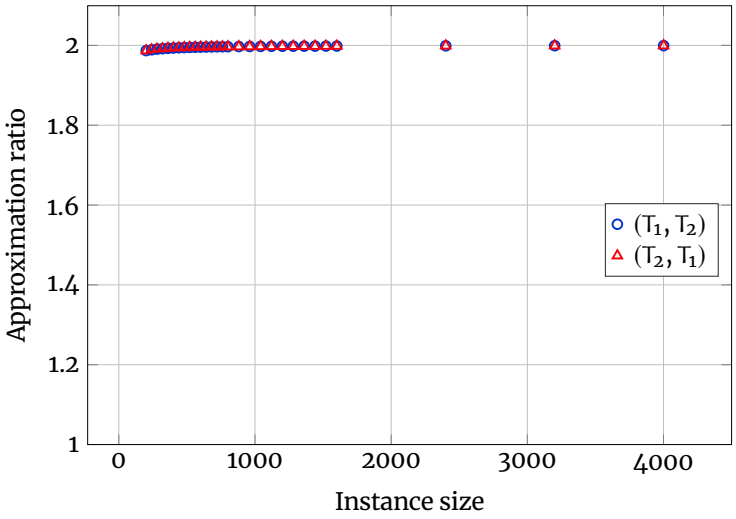


Figure 5.5: The approximation ratio and the running time (s) of WZ on HUGE.



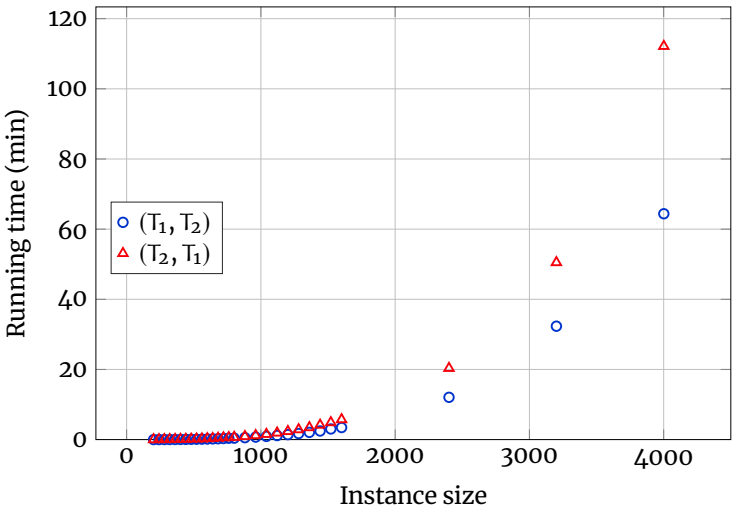
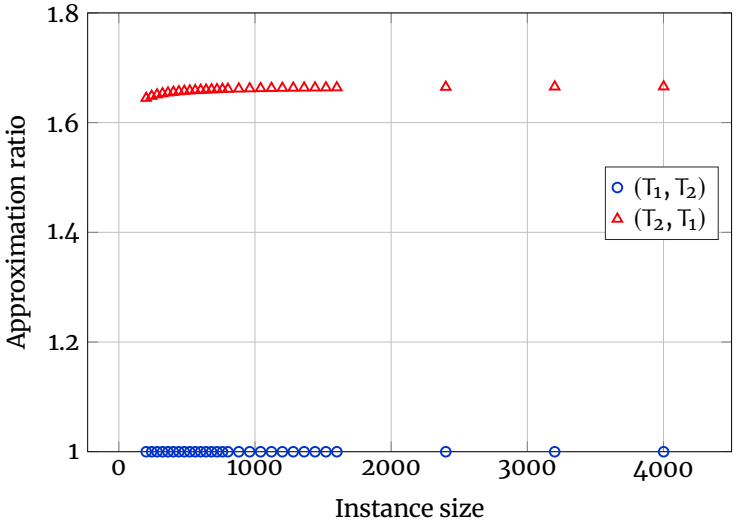


Figure 5.6: The approximation ratio and the running time (min) of SNV on HUGE.



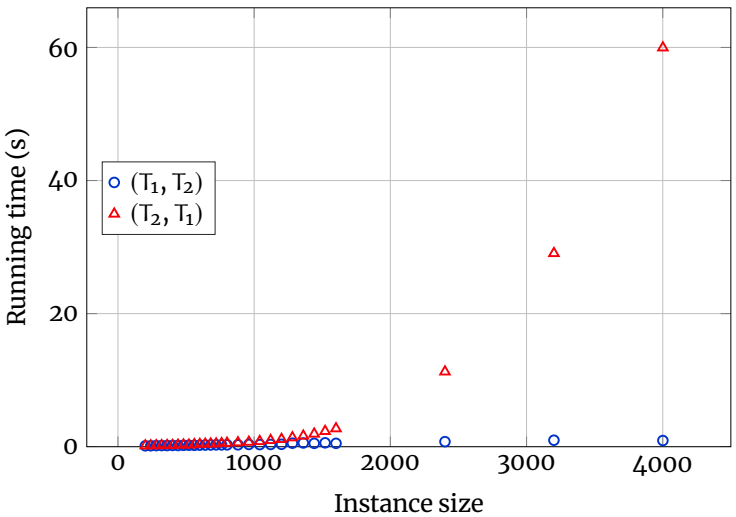
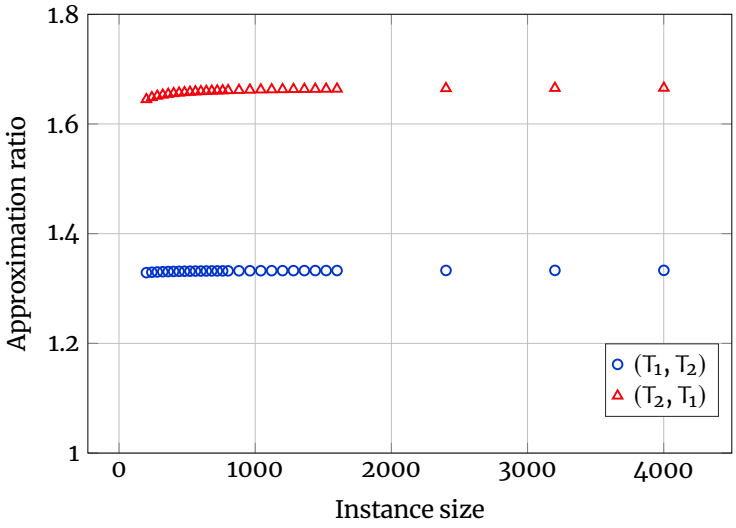


Figure 5.7: The approximation ratio and the running time (s) of CMW on HUGE.



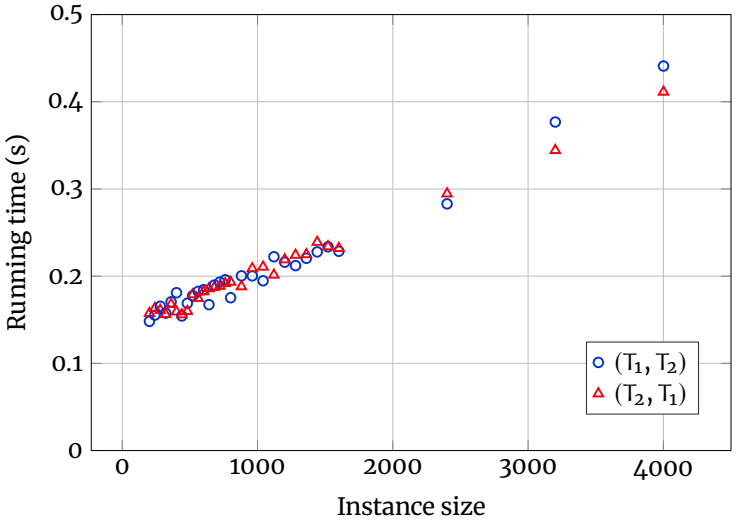
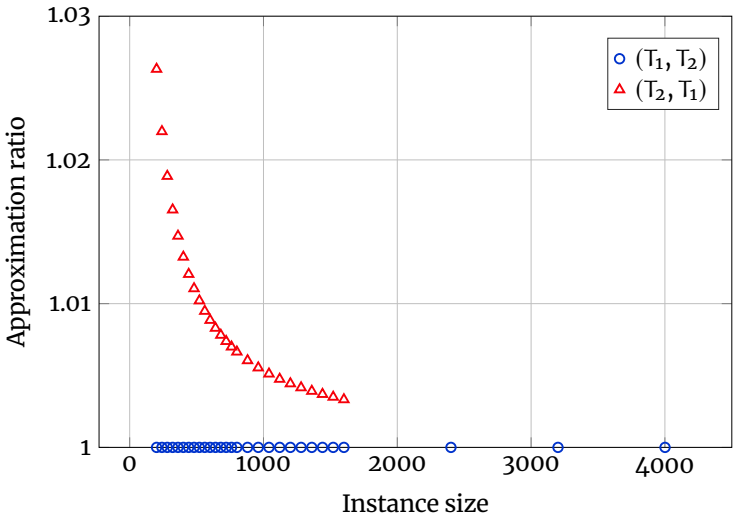


Figure 5.8: The approximation ratio and the running time (s) of CHN on HUGE.



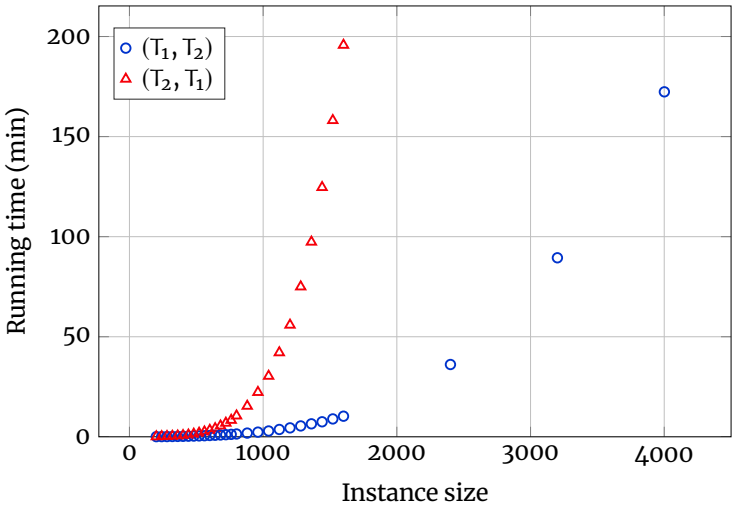
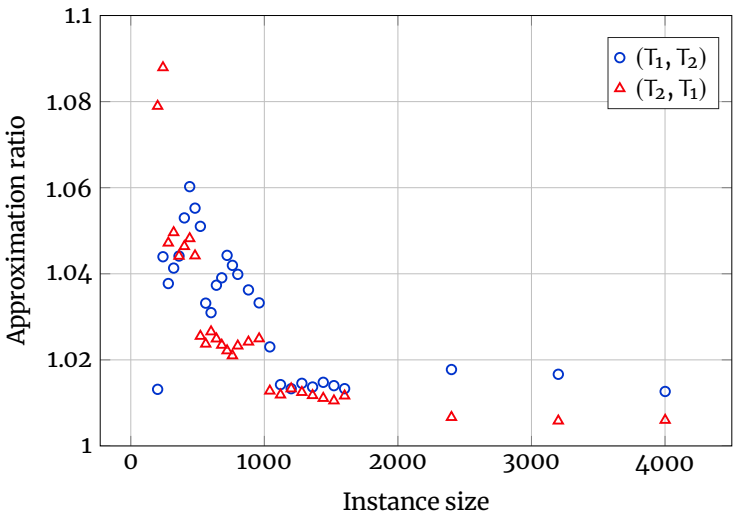


Figure 5.9: The approximation ratio and the running time (min) of CombMCTS on HUGE. Three largest instances are missing as the time limit of 24 hrs was exceeded.



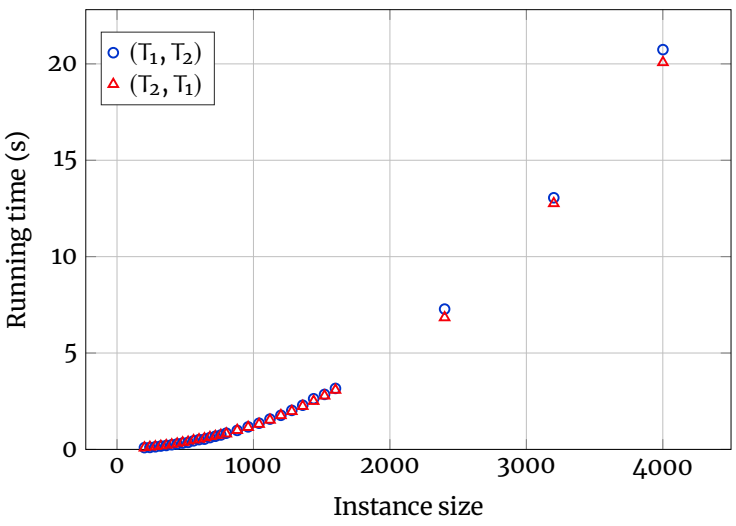


Figure 5.10: The approximation ratio and the running time (s) of NewCCS on HUGE.

	Approximation ratio (T_1, T_2)	**Approximation ratio** (T_2, T_1)
WZ	converges to 5/3	converges to 2
CMW	equals to 1	converges to 5/3
CHN	converges to 4/3	converges to 5/3
CombMCTS	equals to 1	converges to 1

Table 5.3: Behaviors of the approximation ratio of each algorithm with increasing instance sizes in the dataset HUGE, for each of the two possible orderings of the input.

Interestingly, apart from SVN and NewCCS, the other four algorithms all exhibit significantly different behaviors in terms of the approximation ratio when the order of the input trees is switched. Such observations are summarized in Table 5.3. For two of those algorithms, namely CMW and CombMCTS, the running time also follows two distinctive growth patterns, i.e., the running time grows more rapidly with the instance size for one of two orderings of the input, and grows more slowly for the other ordering.

Among the six algorithms, WZ and CHN are the fastest but achieve the worst approximation ratios; SVN always produces an approximation ratio near the worst factor 2; the performance of CMW is most sensitive to the ordering of the input. Only CombMCTS and NewCCS consistently produce near optimal solutions (approximation ratio < 1.1) for both orderings of the input. However, CombMCTS can be more than a thousand times slower than NewCCS (e.g., 200 min versus 3 s), making it impractical on this family of instances as soon as the instance size becomes larger than a few hundreds. We have confirmed again the observation that we made in the previous experiment: NewCCS produces results of good quality, is fast enough, and is robust against special cases (no errors, no sensitivity to the ordering of the input).

5.5 Conclusion and perspectives

We proposed a new algorithm NewCCS for finding an approximate solution of the MAXIMUM AGREEMENT FOREST PROBLEM. Although its approximation ratio is currently unknown, it performs well in practice, beating existing approximation algorithms either in terms of the quality of the solution or in terms of speed. The immediate perspective is to prove or disprove Conjecture 5.1. There are a number of small known instances on which NewCCS achieves the factor 2. Based only on few simple ideas, NewCCS is easier to understand and to implement than the two algorithms that achieve the current best approximation ratio of 2, and it has the potential to be improved in the future. Finding a better approximation algorithm for computing the rSPR distance is an interesting question both theoretically and in practical applications. Any improvement in the approximation factor or in the running time can possibly lead to better exact algorithms for computing the rSPR distance (e.g., by using a bounded search technique), as well as better exact or approximation algorithms for related problems such as the computation of the hybridization number.

Conclusion

The interaction between organisms living in closely related environments, such as humans and their gut microbiota, can present complex dynamics. To understand it through the lens of coevolution, the field of computational biology provides powerful methods for finding the most parsimonious reconciliations between phylogenetic trees. In the case where efficient enumeration algorithms exist, that is, when we have put behind us the *computational* complexity of the problem itself, what we are facing now is the *biological* complexity of the solutions of the problem: in a hosts/parasites system consisting of more than a thousand species, any of the current event-based models would generate a solution space that is so large that it precludes the analysis or the interpretation of the results [Bau+14]. There are two natural ways of addressing this issue: either we refine the model by including more constraints, hoping that the output size will be significantly reduced (and hoping that we can still solve the problem efficiently), or, we cling to this huge solution space but try to summarize it somehow, so that we get an output of reasonable size without losing important biological information. This thesis is concerned with the second approach.

In Chapter 2, we defined different notions of equivalence between reconciliations. To summarize the solution space of the phylogenetic tree reconciliation problem then means to provide a list of equivalence classes. In each equivalence class, any solution can be chosen as the representative, as the biological characteristics that we want to capture with the equivalence relation are necessarily shared by any solution in the class. We designed practical algorithms for enumerating the equivalence classes and the representative solutions, and in Chapter 3, applied those algorithms to biological datasets and made them available to end users through the Capybara software.

Our approach is remarkable from several points of view. First of all, for the phylogenetic tree reconciliation problem, the practical benefit of considering the equivalence classes when analyzing the solution space is striking. For example, in Chapter 3, Section 3.1.3, we managed to reduce the number of solutions that need to be considered from the order of 10^{42} to only 7. Beyond the reconciliation problem, many more problems in bioinformatics or computer science also suffer from a large size of the solution space. Compared to other possible approaches that can drastically reduce the size of the output, such as agglomerative clustering methods that produce consensus solutions or centroid solutions [Ozd+17; ML19; San+20; AQE19], the equivalence-based approach has quite a few advantages.

The first one is that, since we only need to define an adequate notion of equivalence, the equivalence-based approach can be used in a wider context than clustering methods. The latter require some kind of distance or diversity measure between solutions which, depending on their mathematical nature (numerical values, strings of symbols, graphs, functions on graphs, etc.), can be difficult to define or costly to compute (see also Chapter 4, Section 4.2).

The second advantage is the interpretability of the results. One reason for this is that the enumeration (and the counting) of equivalence classes, as we formulated it, is not a data analysis method but instead a computational problem that could be solved exactly. Clustering methods will perform poorly when the solution space is not well-separated into clusters with respect to the chosen distance measure, and, when there are indeed some distinctive clusters in the distribution of the solutions, such methods would fail to discover the pattern if the number of "real" clusters is too large (this is because the running time usually depends heavily on the size of the output, i.e., the final number of clusters chosen by the user). Quite unlike the clustering methods, an algorithm that counts or enumerates the equivalence classes will invariably discover any "distinctive pattern" with respect to the chosen equivalence relation, as a pattern here corresponds simply to an equivalence class. The equivalence relation also provides a precise characterization of the solutions inside an equivalence class. In contrast, solutions inside a cluster in the clustering output are described by their similarities or

dissimilarities with the representative solution (the centroid of the cluster): these are numerical values which can be less straightforward for the user to interpret.

For all these reasons, we believe that, outside of the context of the reconciliation problem, there are more problems that can benefit from considering the equivalence classes of solutions. In Chapter 4, we proposed a general framework for enumerating equivalence classes of solutions that can be useful for a variety of problems.

This general framework has a strong connection with the key Algorithm 1 of Chapter 2 that enumerates E-equivalence classes in polynomial delay for the reconciliation problem. First, while Algorithm 1 is described in terms of the reconciliation graph which might sound limited to that particular problem, we pointed out early on that this graph has the same structural properties as ad-AND/OR graphs, a type of graph that naturally emerges in problems that can be solved by dynamic programming algorithms. Moreover, since Algorithm 1 only employed very general techniques such as recursive depth-first search and backtracking, it is not at all surprising that the idea can be generalized to tackle a wider range of problems.

Making sense of the output by efficiently summarizing the solution space is only one way of increasing the practical usability of an algorithm. In Chapter 5, we saw that even for the well-studied problem of computing the rooted subtree prune-and-regraft distance between two phylogenetic trees, current algorithms stumble upon various practical issues: quality of the solutions, speed, or the robustness of the implementation when tested on special datasets (see Section 5.4). We thus proposed a new algorithm that is easy to understand and to implement, is fast enough, and consistently produces solutions of high quality on randomly generated instances as well as on special families of instances.

Other than the open mathematical and computational challenges mentioned at the end of each chapter, there remains several avenues of expansion for the work of the present thesis. It will be interesting to see how well the results of equivalence classes enumeration match up with the biological understanding of the coevolutionary history of the organisms in other datasets, and maybe under new notions of equivalence that yet need to be defined. Then, based on the feedback from end users, new features can

be added to Capybara to include more computational and visualization options. It will also be interesting to apply the general framework of Chapter 4 to study some different problems in order to see how well it works in practice. We provided a few examples that can be a good starting point, and we would like to find more applications. For problems whose solutions do not have a tree structure, the enumeration of equivalence classes as a means of understanding the solution space can still be worth investigating, although new techniques will be necessary.

References

[AQE19] Nuraini Aguse, Yuanyuan Qi, and Mohammed El-Kebir. "Summarizing the solution space in tumor phylogeny inference by multiple consensus trees". In: *Bioinformatics* 35.14 (July 2019), pp. i408–i416. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btz312.

[AL11] Husain Aljazzar and Stefan Leue. "K*: A heuristic search algorithm for finding the k shortest paths". In: *Artificial Intelligence* 175.18 (2011), pp. 2129–2154. ISSN: 0004-3702. DOI: 10.1016/j.artint.2011.07.003.

[AS01] Benjamin L. Allen and Mike Steel. "Subtree Transfer Operations and Their Induced Metrics on Evolutionary Trees". In: *Annals of Combinatorics* 5.1 (June 2001), pp. 1–15. ISSN: 0219-3094. DOI: 10.1007/s00026-001-8006-8.

[ASA19] Mohammed Alokshiya, Saeed Salem, and Fidaa Abed. "A linear delay algorithm for enumerating all connected induced subgraphs". In: *BMC Bioinformatics* 20.12 (June 2019), p. 319. ISSN: 1471-2105. DOI: 10.1186/s12859-019-2837-y.

[AMP97] Steen A. Andersson, David Madigan, and Michael D. Perlman. "A characterization of Markov equivalence classes for acyclic digraphs". In: *Annals of Statistics* 25.2 (Apr. 1997), pp. 505–541. DOI: 10.7916/D8280JSB.

[AK11] Albert Angel and Nick Koudas. "Efficient Diversity-Aware Search". In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. SIGMOD '11. Athens, Greece: Association for Computing Ma-

chinery, 2011, pp. 781–792. ISBN: 9781450306614. DOI: 10.1145/1989323.1989405.

[Bah+16] Simon Bahrndorff, Tibebu Alemu, Temesgen Alemneh, and Jeppe Lund Nielsen. "The Microbiome of Animals: Implications for Conservation Biology". In: *International Journal of Genomics* 2016 (Apr. 2016), p. 5304028. ISSN: 2314-436X. DOI: 10.1155/2016/5304028.

[BMB13] Juan Antonio Balbuena, Raúl Míguez-Lozano, and Isabel Blasco-Costa. "PACo: A Novel Procrustes Application to Cophylogenetic Analysis". In: *PLOS ONE* 8.4 (Apr. 2013), pp. 1–15. DOI: 10.1371/journal.pone.0061048.

[BAK12] Mukul S. Bansal, Eric J. Alm, and Manolis Kellis. "Efficient algorithms for the reconciliation problem with gene duplication, horizontal transfer and loss". In: *Bioinformatics* 28.12 (2012), pp. 283–291. DOI: 10.1093/bioinformatics/bts225.

[BAK13] Mukul S. Bansal, Eric J. Alm, and Manolis Kellis. "Reconciliation revisited: handling multiple optima when reconciling with duplication, transfer, and loss". In: *Journal of computational biology* 20.10 (Oct. 2013), pp. 738–754. ISSN: 1557-8666. DOI: 10.1089/cmb.2013.0073.

[Bar+05] Mihaela Baroni, Stefan Grünewald, Vincent Moulton, and Charles Semple. "Bounding the Number of Hybridisation Events for a Consistent Evolutionary History". In: *Journal of Mathematical Biology* 51.2 (Aug. 2005), pp. 171–182. ISSN: 1432-1416. DOI: 10.1007/s00285-005-0315-9.

[Bau+14] Christian Baudet, Beatrice Donati, Blerina Sinaimeri, Pierluigi Crescenzi, Christian Gautier, Catherine Matias, and Marie-France Sagot. "Cophylogeny Reconstruction via an Approximate Bayesian Computation". In: *Systematic Biology* 64.3 (Dec. 2014), pp. 416–431. ISSN: 1063-5157. DOI: 10.1093/sysbio/syu129.

[BH06] Robert G. Beiko and Nicholas Hamilton. "Phylogenetic identification of lateral genetic transfer events". In: *BMC Evolutionary Biology* 6.1 (Feb. 2006), p. 15. ISSN: 1471-2148. DOI: 10.1186/1471-2148-6-15.

[Bel52] Richard Bellman. "On the Theory of Dynamic Programming". In: *Proceedings of the National Academy of Sciences* 38.8 (1952), pp. 716–719. ISSN: 0027-8424. DOI: 10.1073/pnas.38.8.716.

[Bel62] Richard Bellman. "Dynamic Programming Treatment of the Travelling Salesman Problem". In: *Journal of the ACM* 9.1 (Jan. 1962), pp. 61–63. ISSN: 0004-5411. DOI: 10.1145/321105.321111.

[Bel13] Richard Bellman. *Dynamic Programming*. Dover Books on Computer Science. Dover Publications, 2013. ISBN: 9780486317199.

[BK60] Richard Bellman and Robert Kalaba. "On kth Best Policies". In: *Journal of the Society for Industrial and Applied Mathematics* 8.4 (1960), pp. 582–588. ISSN: 03684245. DOI: 10.1137/0108044.

[BB72] Umberto Bertele and Francesco Brioschi. *Nonserial Dynamic Programming*. USA: Academic Press, Inc., 1972. ISBN: 0120934507.

[Blu+87] Anselm Blumer, Janet A. Blumer, David H. Haussler, Ross M. McConnell, and Andrzej Ehrenfeucht. "Complete Inverted Files for Efficient Text Retrieval and Analysis". In: *Journal of the ACM* 34.3 (July 1987), pp. 578–595. ISSN: 0004-5411. DOI: 10.1145/28869.28873.

[Bod88] Hans L. Bodlaender. "Dynamic programming on graphs with bounded treewidth". In: *Automata, Languages and Programming*. Ed. by Timo Lepistö and Arto Salomaa. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 105–118. ISBN: 978-3-540-39291-0. DOI: 10.1007/3-540-19488-6_110.

[Bon+06] Maria Luisa Bonet, Katherine St. John, Ruchi Mahindru, and Nina Amenta. "Approximating Subtree Distances Between Phylogenies". In: *Journal of Computational Biology* 13.8 (2006), pp. 1419–1434. DOI: 10.1089/cmb.2006.13.1419.

[Bon70] Pierre E. Bonzon. "Necessary and Sufficient Conditions for Dynamic Programming of Combinatorial Type". In: *Journal of the ACM* 17 (1970), pp. 675–682. DOI: 10.1145/321607.321616.

[BMS08] Magnus Bordewich, Catherine McCartin, and Charles Semple. "A 3-approximation algorithm for the subtree distance between phylogenies". In: *Journal of Discrete Algorithms* 6.3 (2008), pp. 458–471. ISSN: 1570-8667. DOI: 10.1016/j.jda.2007.10.002.

[BS05] Magnus Bordewich and Charles Semple. "On the Computational Complexity of the Rooted Subtree Prune and Regraft Distance". In: *Annals of Combinatorics* 8.4 (Jan. 2005), pp. 409–423. ISSN: 0219-3094. DOI: 10.1007/s00026-004-0229-z.

[BS07] Magnus Bordewich and Charles Semple. "Computing the minimum number of hybridization events for a consistent evolutionary history". In: *Discrete Applied Mathematics* 155.8 (2007), pp. 914–928. ISSN: 0166-218X. DOI: 10.1016/j.dam.2006.08.008.

[Bra+08] Marília D.V. Braga, Marie-France Sagot, Celine Scornavacca, and Eric Tannier. "Exploring the Solution Space of Sorting by Reversals, with Experiments and an Application to Evolution". In: *IEEE/ACM transactions on computational biology and bioinformatics* 5 3 (2008), pp. 348–56. DOI: 10.1109/TCBB.2008.16.

[BS10] Marília D.V. Braga and Jens Stoye. "The Solution Space of Sorting by DCJ". In: *Journal of Computational Biology* 17.9 (2010), pp. 1145–1165. DOI: 10.1089/cmb.2010.0109.

[BDI11] Joshua Buresh-Oppenheim, Sashka Davis, and Russell Impagliazzo. "A Stronger Model of Dynamic Programming Algorithms". In: *Algorithmica* 60 (Aug. 2011), pp. 938–968. DOI: 10.1007/s00453-009-9385-1.

[Bur+20] Fabien Burki, Andrew J. Roger, Matthew W. Brown, and Alastair G.B. Simpson. "The New Tree of Eukaryotes". In: *Trends in Ecology & Evolution* 35.1 (Jan. 2020), pp. 43–55. ISSN: 0169-5347. DOI: 10.1016/j.tree.2019.08.008.

[Cha98] Michael A. Charleston. "Jungles: a new solution to the host/parasite phylogeny reconciliation problem". In: *Mathematical Biosciences* 149.2 (1998), pp. 191–223. ISSN: 0025-5564. DOI: 10.1016/S0025-5564(97)10012-8.

[Che+20] Zhi-Zhong Chen, Youta Harada, Yuna Nakamura, and Lusheng Wang. "Faster Exact Computation of rSPR Distance via Better Approximation". In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 17.3 (2020), pp. 916–929. DOI: 10.1109/TCBB.2018.2878731.

[CMW16] Zhi-Zhong Chen, Eita Machida, and Lusheng Wang. "An Improved Approximation Algorithm for rSPR Distance". In: *Computing and Combinatorics (COCOON 2016)*. Ed. by Thang N. Dinh and My T. Thai. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 468–479. ISBN: 978-3-319-42634-1. DOI: 10.1007/978-3-319-42634-1_38.

[CK20] JaeJin Choi and Sung-Hou Kim. "Whole-proteome tree of life suggests a deep burst of organism diversity". In: 117.7 (2020), pp. 3678–3686. ISSN: 0027-8424. DOI: 10.1073/pnas.1915766117.

[Con+10] Chris Conow, Daniel Fielder, Yaniv Ovadia, and Ran Libeskind-Hadas. "Jane: a new tool for the cophylogeny reconstruction problem". In: *Algorithms for molecular biology* 5 (Feb. 2010), pp. 16–16. ISSN: 1748-7188. DOI: 10.1186/1748-7188-5-16.

[Dar88] Charles Darwin. *The Origins of Species by Means of Natural Selection, Or the Preservation of Favoured Races in the Struggle for Life*. J. Murray, 1888. URL: https://books.google.fr/books?id=YVQ5ynn6mpUC.

[DA11] Lawrence A. David and Eric J. Alm. "Rapid evolutionary innovation during an Archaean genetic expansion". In: *Nature* 469.7328 (Jan. 2011), pp. 93–96. ISSN: 1476-4687. DOI: 10.1038/nature09649.

[Day85] William H. E. Day. "Optimal algorithms for comparing trees with labeled leaves". In: *Journal of Classification* 2.1 (Dec. 1985), pp. 7–28. ISSN: 1432-1343. DOI: 10.1007/BF01908061.

[DM07] Rina Dechter and Robert Mateescu. "AND/OR search spaces for graphical models". In: *Artificial Intelligence* 171.2 (2007), pp. 73–106. ISSN: 0004-3702. DOI: 10.1016/j.artint.2006.11.003.

[DBP05] Frédéric Delsuc, Henner Brinkmann, and Hervé Philippe. "Phylogenomics and the reconstruction of the tree of life". In: *Nature Reviews Genetics* 6.5 (May 2005), pp. 361–375. ISSN: 1471-0064. DOI: 10.1038/nrg1603.

[Den+13] Jun Deng, Fang Yu, Hai-Bin Li, Marco Gebiola, Yves Desdevises, San-An Wu, and Yan-Zhou Zhang. "Cophylogenetic relationships between *Anicetus* parasitoids (Hymenoptera: Encyrtidae) and their scale insect hosts (Hemiptera: Coccidae)". In: *BMC Evolutionary Biology* 13.1 (2013), p. 275. DOI: 10.1186/1471-2148-13-275.

[Die05] Reinhard Diestel. *Graph Theory (Graduate Texts in Mathematics)*. Springer, Aug. 2005. ISBN: 3540261826.

[Don+15] Beatrice Donati, Christian Baudet, Blerina Sinaimeri, Pierluigi Crescenzi, and Marie-France Sagot. "Eucalypt: Efficient tree reconciliation enumerator". In: *Algorithms for Molecular Biology* 10.1 (2015), p. 3. DOI: 10.1186/s13015-014-0031-3.

[Doy+10] Jean-Philippe Doyon, Celine Scornavacca, K. Yu. Gorbunov, Gergely J. Szöllősi, Vincent Ranwez, and Vincent Berry. "An Efficient Algorithm for Gene/Species Trees Parsimonious Reconciliation with Losses, Duplications and Transfers". In: *Comparative Genomics (RECOMB-CG 2010)*. Ed. by Eric Tannier. Vol. 6398. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 93–108. ISBN: 978-3-642-16181-0. DOI: 10.1007/978-3-642-16181-0_9.

[Epp98] David Eppstein. "Finding the k Shortest Paths". In: *SIAM Journal on Computing* 28.2 (1998), pp. 652–673. DOI: 10.1137/S0097539795290477.

[EKM16] Sylvie Estrela, Benjamin Kerr, and Jeffrey J. Morris. "Transitions in individuality through symbiosis". In: *Current Opinion in Microbiology* 31 (2016).

Environmental microbiology * Special Section: Megaviromes, pp. 191–198. ISSN: 1369-5274. DOI: 10.1016/j.mib.2016.04.007.

[Eth+06] Graham J. Etherington, Susan M. Ring, Michael A. Charleston, Jo Dicks, Vic J. Rayward-Smith, and Ian N. Roberts. "Tracing the origin and co-phylogeny of the caliciviruses". In: *Journal of General Virology* 87.5 (2006), pp. 1229–1235. DOI: 10.1099/vir.0.81635-0.

[FH05] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. "Pictorial Structures for Object Recognition". In: *International Journal of Computer Vision* 61.1 (Jan. 2005), pp. 55–79. ISSN: 0920-5691. DOI: 10.1023/B:VISI.0000042934.15159.49.

[FGS19] Henning Fernau, Petr A. Golovach, and Marie-France Sagot. "Algorithmic Enumeration: Output-sensitive, Input-Sensitive, Parameterized, Approximative (Dagstuhl Seminar 18421)". In: *Dagstuhl Reports* 8.10 (2019). Ed. by Henning Fernau, Petr A. Golovach, and Marie-France Sagot, pp. 63–86. ISSN: 2192-5283. DOI: 10.4230/DagRep.8.10.63.

[Fit70] Walter M. Fitch. "Distinguishing Homologous from Analogous Proteins". In: *Systematic Biology* 19.2 (June 1970), pp. 99–113. ISSN: 1063-5157. DOI: 10.2307/2412448.

[FJ84] Greg N. Frederickson and Donald B. Johnson. "Generalized Selection and Ranking: Sorted Matrices". In: *SIAM Journal on Computing* 13.1 (1984), pp. 14–30. DOI: 10.1137/0213002.

[Geo+09] Elisabeth Georgii, Sabine Dietmann, Takeaki Uno, Philipp Pagel, and Koji Tsuda. "Enumeration of condition-dependent dense modules in protein interaction networks". In: *Bioinformatics* 25.7 (Feb. 2009), pp. 933–940. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btp080.

[Gho+12] Priyankar Ghosh, Amit Sharma, P. P. Chakrabarti, and Pallab Dasgupta. "Algorithms for Generating Ordered Solutions for Explicit AND/OR Structures: extended abstract". In: *Journal of Artificial Intelligence Research* 44 (2012), pp. 275–333. DOI: 10.1613/jair.3576.

[GVR04] Robert Giegerich, Björn Voß, and Marc Rehmsmeier. "Abstract shapes of RNA". In: *Nucleic Acids Research* 32.16 (Jan. 2004), pp. 4843–4851. ISSN: 0305-1048. DOI: 10.1093/nar/gkh779.

[Glu63] Brian Gluss. "A Method for Obtaining Suboptimal Group-Testing Policies Using Dynamic Programming and Information Theory". In: *Journal of the ACM* 10.1 (Jan. 1963), pp. 89–96. ISSN: 0004-5411. DOI: 10.1145/321150.321156.

[GMM81] Stefania Gnesi, Ugo Montanari, and Alberto Martelli. "Dynamic Programming as Graph Searching: An Algebraic Approach". In: *Journal of the ACM* 28.4 (Aug. 1981), pp. 737–751. ISSN: 0004-5411. DOI: 10.1145/322276.322285.

[Haa+19] Jordan Haack, Eli Zupke, Andrew Ramirez, Yi-Chieh Wu, and Ran Libeskind-Hadas. "Computing the Diameter of the Space of Maximum Parsimony Reconciliations in the Duplication-Transfer-Loss Model". In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 16.1 (2019), pp. 14–22. DOI: 10.1109/TCBB.2018.2849732.

[HN88] Mark S. Hafner and Steven A. Nadler. "Phylogenetic trees support the coevolution of parasites and their hosts". In: *Nature* 332.6161 (Mar. 1988), pp. 258–259. ISSN: 1476-4687. DOI: 10.1038/332258a0.

[HP95] Mark S. Hafner and Roderic D. M. Page. "Molecular Phylogenies and Host-Parasite Cospeciation: Gophers and Lice as a Model System". In: *Philosophical Transactions: Biological Sciences* 349.1327 (1995), pp. 77–83. ISSN: 09628436. DOI: 10.1098/rstb.1995.0093.

[Hal80] William K. Hale. "Frequency assignment: Theory and applications". In: *Proceedings of the IEEE* 68.12 (1980), pp. 1497–1514. DOI: 10.1109/PROC.1980.11899.

[Hei+96] Jotun Hein, Tao Jiang, Lusheng Wang, and Kaizhong Zhang. "On the complexity of comparing evolutionary trees". In: *Discrete Applied Mathematics*

71.1 (1996), pp. 153–169. ISSN: 0166-218X. DOI: 10.1016/S0166-218X(96)00062-5.

[HK62] Michael Held and Richard M. Karp. "A Dynamic Programming Approach to Sequencing Problems". In: *Journal of the Society for Industrial and Applied Mathematics* 10.1 (1962), pp. 196–210. DOI: 10.1137/0110015.

[Hel89] Paul Helman. "A Common Schema for Dynamic Programming and Branch and Bound Algorithms". In: *Journal of the ACM* 36.1 (Jan. 1989), pp. 97–128. ISSN: 0004-5411. DOI: 10.1145/58562.59304.

[HMS07] John Hershberger, Matthew Maxel, and Subhash Suri. "Finding the k Shortest Simple Paths: A New Algorithm and Its Implementation". In: *ACM Transactions on Algorithms* 3.4 (Nov. 2007), 45–es. ISSN: 1549-6325. DOI: 10.1145/1290672.1290682.

[HS93] Liisa Holm and Chris Sander. "Protein Structure Comparison by Alignment of Distance Matrices". In: *Journal of Molecular Biology* 233.1 (1993), pp. 123–138. ISSN: 0022-2836. DOI: 10.1006/jmbi.1993.1489.

[Hug+16] Laura A. Hug et al. "A new view of the tree of life". In: *Nature Microbiology* 1.5 (Apr. 2016), p. 16048. ISSN: 2058-5276. DOI: 10.1038/nmicrobiol.2016.48.

[Hug+07] Joseph Hughes, Martyn Kennedy, Kevin P. Johnson, Ricardo L. Palma, and Roderic D. M. Page. "Multiple Cophylogenetic Analyses Reveal Frequent Cospeciation between Pelecaniform Birds and *Pectinopygus* Lice". In: *Systematic Biology* 56.2 (Apr. 2007), pp. 232–251. ISSN: 1063-5157. DOI: 10.1080/10635150701311370.

[Hug99] Jean-Pierre Hugot. "Primates and Their Pinworm Parasites: The Cameron Hypothesis Revisited". In: *Systematic Biology* 48.3 (July 1999), pp. 523–546. ISSN: 1063-5157. DOI: 10.1080/106351599260120.

[Hug03] Jean-Pierre Hugot. "New evidence of Hystricognath Rodents monophyly from the phylogeny of their pinworms". In: *Tangled Trees: Phylogeny, Cospe-*

ciation, and Coevolution. Ed. by Roderic D. M. Page. Chicago, USA: UC Press, Jan. 2003. Chap. 6, pp. 144–173. ISBN: 0226644677.

[HRS10] Daniel H. Huson, GermanyRegula Rupp, and Celine Scornavacca. *Phylogenetic Networks: Concepts, Algorithms and Applications*. Cambridge University Press, Dec. 2010. ISBN: 9780521755962.

[Iba74] Toshihide Ibaraki. "Classes of discrete optimization problems and their decision problems". In: *Journal of Computer and System Sciences* 8.1 (1974), pp. 84–116. ISSN: 0022-0000. DOI: 10.1016/S0022-0000(74)80024-3.

[JT00] Pablo Jiménez and Carme Torras. "An efficient algorithm for searching implicit AND/OR graphs with cycles". In: *Artificial Intelligence* 124.1 (2000), pp. 1–30. ISSN: 0004-3702. DOI: 10.1016/S0004-3702(00)00063-1.

[KYT20] Paschalia Kapli, Ziheng Yang, and Maximilian J. Telford. "Phylogenetic tree building in the genomic age". In: *Nature Reviews Genetics* 21.7 (July 2020), pp. 428–444. ISSN: 1471-0064. DOI: 10.1038/s41576-020-0233-0.

[KH67] Richard M. Karp and Michael Held. "Finite-State Processes and Dynamic Programming". In: *SIAM Journal on Applied Mathematics* 15.3 (1967), pp. 693–718. DOI: 10.1137/0115060.

[KIM82] Naoki Katoh, Toshihide Ibaraki, and Hisashi Mine. "An efficient algorithm for K shortest simple paths". In: *Networks* 12.4 (1982), pp. 411–427. DOI: 10.1002/net.3230120406.

[KPP04] Hans Kellerer, Ulrich Pferschy, and David Pisinger. "Basic Algorithmic Concepts". In: *Knapsack Problems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 15–42. ISBN: 978-3-540-24777-7. DOI: 10.1007/978-3-540-24777-7_2.

[LO14] Bonnie R. Lei and Kevin J. Olival. "Contrasting Patterns in Mammal-Bacteria Coevolution: Bartonella and Leptospira in Bats and Rodents". In: *PLOS Neglected Tropical Diseases* 8.3 (Mar. 2014), pp. 1–11. DOI: 10.1371/journal.pntd.0002738.

[LP08] Tommy L. F. Leung and Robert Poulin. "Parasitism, commensalism, and mutualism: exploring the many shades of symbioses". In: *Vie et Milieu / Life & Environment* 58.2 (2008), pp. 107–115. ISSN: 0240-8759. URL: https://hdl.handle.net/1959.11/8871.

[Ley+08] Ruth E. Ley et al. "Evolution of Mammals and Their Gut Microbes". In: *Science* 320.5883 (2008), pp. 1647–1651. ISSN: 0036-8075. DOI: 10.1126/science.1155725.

[Ma+18] Weiyun Ma, Dmitriy Smirnov, Juliet Forman, Annalise Schweickart, Carter Slocum, Srinidhi Srinivasan, and Ran Libeskind-Hadas. "DTL-RnB: Algorithms and Tools for Summarizing the Space of DTL Reconciliations". In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 15.2 (2018), pp. 411–421. DOI: 10.1109/TCBB.2016.2537319.

[MB85] Ambuj Mahanti and Amitava Bagchi. "AND/OR Graph Heuristic Search Methods". In: *Journal of the ACM* 32.1 (Jan. 1985), pp. 28–51. ISSN: 0004-5411. DOI: 10.1145/2455.2459.

[MM78] Alberto Martelli and Ugo Montanari. "Optimizing Decision Trees through Heuristically Guided Search". In: *Communications of the ACM* 21.12 (Dec. 1978), pp. 1025–1039. ISSN: 0001-0782. DOI: 10.1145/359657.359664.

[ML19] Ross Mawhorter and Ran Libeskind-Hadas. "Hierarchical clustering of maximum parsimony reconciliations". In: *BMC Bioinformatics* 20 (2019), p. 612. DOI: 10.1186/s12859-019-3223-5.

[MM05] Daniel Merkle and Martin Middendorf. "Reconstruction of the cophylogenetic history of related phylogenetic trees with divergence timing information". In: *Theory in Biosciences* 123.4 (Apr. 2005), pp. 277–299. ISSN: 1611-7530. DOI: 10.1016/j.thbio.2005.01.003.

[MSS14] Cristian Molinaro, Amy Sliva, and Vs S. Subrahmanian. "Super-Solutions: Succinctly Representing Solutions in Abductive Annotated Probabilistic Temporal Logic". In: *ACM Transactions on Computational Logic* 15.3 (July 2014). ISSN: 1529-3785. DOI: 10.1145/2627354.

[Mor15] Katherine Morrison. "An enumeration of the equivalence classes of self-dual matrix codes". In: *Advances in Mathematics of Communications* 9 (2015), p. 415. ISSN: 1930-5346. DOI: 10.3934/amc.2015.9.415.

[Mou+14] Pierre-Nicolas Mougel, Christophe Rigotti, Marc Plantevit, and Olivier Gandrillon. "Finding maximal homogeneous clique sets". In: *Knowledge and Information Systems* 39.3 (June 2014), pp. 579–608. ISSN: 0219-3116. DOI: 10.1007/s10115-013-0625-y.

[Nar+07] Kazuyuki Narisawa, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. "Efficient Computation of Substring Equivalence Classes with Suffix Arrays". In: *Combinatorial Pattern Matching (CPM 2007)*. Ed. by Bin Ma and Kaizhong Zhang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 340–351. DOI: 10.1007/s00453-016-0178-z.

[NW70] Saul B. Needleman and Christian D. Wunsch. "A general method applicable to the search for similarities in the amino acid sequence of two proteins". In: *Journal of Molecular Biology* 48.3 (1970), pp. 443–453. ISSN: 0022-2836. DOI: 10.1016/0022-2836(70)90057-4.

[Ngu+13] Thi-Hau Nguyen, Vincent Ranwez, Vincent Berry, and Celine Scornavacca. "Support Measures to Estimate the Reliability of Evolutionary Events Predicted by Reconciliation Methods". In: *PLOS ONE* 8.10 (Oct. 2013), pp. 1–14. DOI: 10.1371/journal.pone.0073667.

[Nil82] Nils J. Nilsson. *Principles of Artificial Intelligence*. Tioga, Palo Alto, CA: Springer-Verlag Berlin Heidelberg, 1982. ISBN: 978-3-540-11340-9.

[Nøj+18] Nikolai Nøjgaard, Manuela Geiß, Daniel Merkle, Peter F. Stadler, Nicolas Wieseke, and Marc Hellmuth. "Time-consistent reconciliation maps and forbidden time travel". In: *Algorithms for Molecular Biology* 13.1 (Feb. 2018), p. 2. ISSN: 1748-7188. DOI: 10.1186/s13015-018-0121-8.

[Ova+11] Yaniv J. Ovadia, Daniel Fielder, Chris Conow, and Ran Libeskind-Hadas. "The Cophylogeny Reconstruction Problem Is NP-Complete". In: *Journal*

of Computational Biology 18.1 (2011), pp. 59–65. DOI: 10.1089/cmb.2009.0240.

[Ozd+17] Alex Ozdemir, Michael Sheely, Daniel Bork, Ricson Cheng, Reyna Hulett, Jean Sung, Jincheng Wang, and Ran Libeskind-Hadas. "Clustering the Space of Maximum Parsimony Reconciliations in the Duplication-Transfer-Loss Model". In: *Algorithms for Computational Biology - 4th International Conference (AlCoB 2017)*. Ed. by Daniel Figueiredo, Carlos Martín-Vide, Diogo Pratas, and Miguel A. Vega-Rodríguez. Aveiro, Portugal: Springer International Publishing, June 2017, pp. 127–139. DOI: 10.1007/978-3-319-58163-7_9.

[Pag94] Roderic D.M. Page. "Parallel Phylogenies: Reconstructing the History of Host-Parasite Assemblages". In: *Cladistics* 10.2 (1994), pp. 155–173. ISSN: 0748-3007. DOI: 10.1006/clad.1994.1010.

[PPG03] Adrian Paterson, Ricardo Palma, and Russell Gray. "Drowning on arrival, missing the boat, and x-events: how likely are sorting events". In: *Tangled Trees: Phylogeny, Cospeciation, and Coevolution*. Ed. by Roderic D. M. Page. Chicago, USA: UC Press, Jan. 2003. Chap. 12, pp. 287–307. ISBN: 0226644677.

[Pen+15] Pamela M. Pennington, Louisa Alexandra Messenger, Jeffrey Reina, José G. Juárez, Gena G. Lawrence, Ellen M. Dotson, Martin S. Llewellyn, and Celia Cordón-Rosales. "The Chagas disease domestic transmission cycle in Guatemala: Parasite-vector switches and lack of mitochondrial co-diversification between Triatoma dimidiata and Trypanosoma cruzi subpopulations suggest non-vectorial parasite dispersal across the Motagua valley". In: *Acta Tropica* 151 (2015). Ecology and diversity of Trypanosoma cruzi, pp. 80–87. DOI: 10.1016/j.actatropica.2015.07.014.

[Pou11] Robert Poulin. *Evolutionary Ecology of Parasites*. Princeton University Press, 2011. ISBN: 9781400840809. DOI: 10.1515/9781400840809.

[RHC09] Cadhla Ramsden, Edward C. Holmes, and Michael A. Charleston. "Hantavirus Evolution in Relation to Its Rodent and Insectivore Hosts: No Evidence for Codivergence". In: *Molecular Biology and Evolution* 26.1 (Jan. 2009), pp. 143–153. ISSN: 0737-4038. DOI: 10.1093/molbev/msn234.

[Ref+08] Guislaine Refrégier, Mickaël Le Gac, Florian Jabbour, Alex Widmer, Jacqui A. Shykoff, Roxana Yockteng, Michael E. Hood, and Tatiana Giraud. "Cophylogeny of the anther smut fungi and their caryophyllaceous hosts: Prevalence of host shifts and importance of delimiting parasite species for inferring cospeciation". In: *BMC Evolutionary Biology* 8.1 (Mar. 2008), p. 100. ISSN: 1471-2148. DOI: 10.1186/1471-2148-8-100.

[RE99] Elena Rivas and Sean R. Eddy. "A dynamic programming algorithm for RNA structure prediction including pseudoknots11Edited by I. Tinoco". In: *Journal of Molecular Biology* 285.5 (1999), pp. 2053–2068. ISSN: 0022-2836. DOI: 10.1006/jmbi.1998.2436.

[Rob91] Fred S. Roberts. "T-colorings of graphs: recent results and open problems". In: *Discrete Mathematics* 93.2 (1991), pp. 229–245. ISSN: 0012-365X. DOI: 10.1016/0012-365X(91)90258-4.

[RF81] D.F. Robinson and L.R. Foulds. "Comparison of phylogenetic trees". In: *Mathematical Biosciences* 53.1 (1981), pp. 131–147. ISSN: 0025-5564. DOI: 10.1016/0025-5564(81)90043-2.

[Rod03] Estela M. Rodrigues. "Algoritmos para Comparação de Árvores Filogenéticas e o Problema dos Pontos de Recombinação". PhD thesis. Brazil: University of São Paulo, 2003. Chap. 7. URL: http://www.ime.usp.br/~estela/studies/tese-traducao-cp7.ps.gz.

[RSW07] Estela M. Rodrigues, Marie-France Sagot, and Yoshiko Wakabayashi. "The maximum agreement forest problem: Approximation algorithms and computational experiments". In: *Theoretical Computer Science* 374.1 (2007), pp. 91–110. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2006.12.011.

[RM08] Lior Rokach and Oded Z. Maimon. *Data Mining with Decision Trees: Theory and Applications*. Series in machine perception and artificial intelligence. World Scientific, 2008. ISBN: 978-981-4590-07-5. DOI: 10.1142/9097.

[Rot19] Gunter Rote. "The Maximum Number of Minimal Dominating Sets in a Tree". In: *Proceedings of the 2019 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2019, pp. 1201–1214. DOI: 10.1137/1.9781611975482.73.

[San75] David Sankoff. "Minimal Mutation Trees of Sequences". In: *SIAM Journal on Applied Mathematics* 28.1 (1975), pp. 35–42. DOI: 10.1137/0128004.

[SML19] Santi Santichaivekin, Ross Mawhorter, and Ran Libeskind-Hadas. "An efficient exact algorithm for computing all pairwise distances between reconciliations in the duplication-transfer-loss model". In: *BMC Bioinformatics* 20.20 (Dec. 2019), p. 636. ISSN: 1471-2105. DOI: 10.1186/s12859-019-3203-9.

[San+20] Santi Santichaivekin, Qing Yang, Jingyi Liu, Ross Mawhorter, Justin Jiang, Trenton Wesley, Yi-Chieh Wu, and Ran Libeskind-Hadas. "eMPRess: a systematic cophylogeny reconciliation tool". In: *Bioinformatics* (Nov. 2020). DOI: 10.1093/bioinformatics/btaa978.

[SZS16] Frans Schalekamp, Anke van Zuylen, and Suzanne van der Ster. "A Duality Based 2-Approximation Algorithm for Maximum Agreement Forest". In: *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*. Ed. by Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi. Vol. 55. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 70:1–70:14. ISBN: 978-3-95977-013-2. DOI: 10.4230/LIPIcs.ICALP.2016.70.

[Sco+13] Celine Scornavacca, Wojciech Paprotny, Vincent Berry, and Vincent Ranwez. "Representing a set of reconciliations in a compact way". In: *Journal*

of Bioinformatics and Computational Biology 11.02 (2013), p. 1250025. DOI: 10.1142/S0219720012500254.

[Shi+16] Feng Shi, Qilong Feng, Jie You, and Jianxin Wang. "Improved approximation algorithm for maximum agreement forest of two rooted binary phylogenetic trees". In: *Journal of Combinatorial Optimization* 32.1 (July 2016), pp. 111–143. ISSN: 1573-2886. DOI: 10.1007/s10878-015-9921-7.

[Sim12] Patrícia M. Simões. "Diversity and dynamics of *Wolbachia*-host associations in arthropods from the Society archipelago, French Polynesia". PhD thesis. University of Lyon 1, France, 2012. URL: https://tel.archives-ouvertes.fr/tel-00850707/file/SimoesP2012.pdf.

[Sim+11] Patrícia M. Simões, Gladys Mialdea, Daphné Reiss, Marie-France Sagot, and Sylvain Charlat. "*Wolbachia* detection: an assessment of standard PCR protocols". In: *Molecular Ecology Resources* 11.3 (2011), pp. 567–572. DOI: 10.1111/j.1755-0998.2010.02955.x.

[Sto+12] Maureen Stolzer, Han Lai, Minli Xu, Deepa Sathaye, Benjamin Vernot, and Dannie Durand. "Inferring duplications, losses, transfers and incomplete lineage sorting with nonbinary species trees". In: *Bioinformatics* 28.18 (Sept. 2012), pp. i409–i415. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bts386.

[Tat+00] Roman L. Tatusov, Michael Y. Galperin, Darren A. Natale, and Eugene V. Koonin. "The COG database: a tool for genome-scale analysis of protein functions and evolution". In: *Nucleic Acids Research* 28.1 (2000), pp. 33–36. DOI: 10.1093/nar/28.1.33.

[Tes93] Barry A. Tesman. "List T-colorings of graphs". In: *Discrete Applied Mathematics* 45.3 (1993), pp. 277–289. ISSN: 0166-218X. DOI: 10.1016/0166-218X(93)90015-G.

[THL11] Ali Tofigh, Michael Hallett, and Jens Lagergren. "Simultaneous Identification of Duplications and Lateral Gene Transfers". In: *IEEE/ACM Trans-*

actions on Computational Biology and Bioinformatics 8.2 (2011), pp. 517–535. DOI: 10.1109/TCBB.2010.14.

[Vek05] Olga Veksler. "Stereo correspondence by dynamic programming on a tree". In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*. Vol. 2. 2005, pp. 384–390. DOI: 10.1109/CVPR.2005.334.

[WF74] Robert A. Wagner and Michael J. Fischer. "The String-to-String Correction Problem". In: *Journal of the ACM* 21.1 (Jan. 1974), pp. 168–173. ISSN: 0004-5411. DOI: 10.1145/321796.321811.

[WJ94] Lusheng Wang and Tao Jiang. "On the Complexity of Multiple Sequence Alignment". In: *Journal of Computational Biology* 1.4 (1994), pp. 337–348. DOI: 10.1089/cmb.1994.1.337.

[Wan+20] Yishu Wang, Arnaud Mary, Marie-France Sagot, and Blerina Sinaimeri. "Capybara: equivalence ClAss enumeration of coPhylogenY event-BAsed ReconciliAtions". In: *Bioinformatics* 36.14 (2020), pp. 4197–4199. DOI: 10.1093/bioinformatics/btaa498.

[Wan+21a] Yishu Wang, Arnaud Mary, Marie-France Sagot, and Blerina Sinaimeri. "A general framework for enumerating equivalence classes of solutions". In: *29th Annual European Symposium on Algorithms (ESA 2021)*. Ed. by Rasmus Pagh Petra Mutzel and Grzegorz Herman. Vol. 204. Leibniz International Proceedings in Informatics (LIPIcs) 28. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2021, 28:1–28:14. DOI: 10.4230/LIPIcs.ESA.2021.28.

[Wan+21b] Yishu Wang, Arnaud Mary, Marie-France Sagot, and Blerina Sinaimeri. "Making sense of a cophylogeny output: Efficient listing of representative reconciliations". In: *21th International Workshop on Algorithms in Bioinformatics (WABI 2021)*. Ed. by Alessandra Carbone and Mohammed El-Kebir. Vol. 201. Leibniz International Proceedings in Informatics (LIPIcs) 3. Dagstuhl,

Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2021, 3:1–3:18. DOI: 10.4230/LIPIcs.WABI.2021.3.

[War95] H. Todd Wareham. "A Simplified Proof of the NP- and MAX SNP-Hardness of Multiple Sequence Tree Alignment". In: *Journal of Computational Biology* 2.4 (1995), pp. 509–514. DOI: 10.1089/cmb.1995.2.509.

[WB85] Michael S. Waterman and Thomas H. Byers. "A dynamic programming algorithm to find all solutions in a neighborhood of the optimum". In: *Mathematical Biosciences* 77.1 (1985), pp. 179–188. ISSN: 0025-5564. DOI: 10.1016/0025-5564(85)90096-3.

[WBZ13] Chris Whidden, Robert G. Beiko, and Norbert Zeh. "Fixed-Parameter Algorithms for Maximum Agreement Forests". In: *SIAM Journal on Computing* 42.4 (2013), pp. 1431–1466. DOI: 10.1137/110845045.

[WZ09] Chris Whidden and Norbert Zeh. "A Unifying View on Approximation and FPT of Agreement Forests". In: *Algorithms in Bioinformatics (WABI 2009)*. Ed. by Steven L. Salzberg and Tandy Warnow. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 390–402. ISBN: 978-3-642-04241-6. DOI: 10.1007/978-3-642-04241-6_32.

[Wu08] Yufeng Wu. "A practical method for exact computation of subtree prune and regraft distance". In: *Bioinformatics* 25.2 (Nov. 2008), pp. 190–196. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btn606.

[YCW19] Kohei Yamada, Zhi-Zhong Chen, and Lusheng Wang. "Better Practical Algorithms for rSPR Distance and Hybridization Number". In: *19th International Workshop on Algorithms in Bioinformatics (WABI 2019)*. Ed. by Katharina T. Huber and Dan Gusfield. Vol. 143. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 5:1–5:12. ISBN: 978-3-95977-123-8. DOI: 10.4230/LIPIcs.WABI.2019.5.

[Yen71] Jin Y. Yen. "Finding the K Shortest Loopless Paths in a Network". In: *Management Science* 17.11 (1971), pp. 712–716. DOI: 10.1287/mnsc.17.11.712.

[Yu+17] Guangchuang Yu, David K. Smith, Huachen Zhu, Yi Guan, and Tommy Tsan-Yuk Lam. "ggtree: an r package for visualization and annotation of phylogenetic trees with their covariates and other associated data". In: *Methods in Ecology and Evolution* 8.1 (2017), pp. 28–36. DOI: 10.1111/2041-210X.12628.

[Zen+06] Zhiping Zeng, Jianyong Wang, Lizhu Zhou, and George Karypis. "Coherent Closed Quasi-Clique Discovery from Large Dense Graph Databases". In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Philadelphia, PA, USA: Association for Computing Machinery, 2006, pp. 797–802. ISBN: 1595933395. DOI: 10.1145/1150402.1150506.