Université Clermont Auvergne

École doctorale Sciences Pour l'Ingénieur

Thèse présentée par

Caroline Brosse

pour obtenir le grade de

Docteur d'Université

Spécialité Informatique

Efficient enumeration algorithms for minimal graph completions and deletions

---

Dirigée par **Vincent Limouzy**

Co-encadrée par **Aurélie Lagoutte** et **Lucas Pastor**

Soutenue publiquement le 11 Septembre 2023 devant le jury composé de :

Fatiha Bendali *Maîtresse de conférences (HDR), Université Clermont Auvergne*	Examinatrice
Marthe Bonamy *Chargée de recherche, CNRS, Université de Bordeaux*	Examinatrice
Nofar Carmeli *Chargée de recherche, Inria, Université de Montpellier*	Examinatrice
Roberto Grossi *Professeur, Università di Pisa*	Rapporteur
Christian Laforest *Professeur, Université Clermont Auvergne*	Examinateur
Vincent Limouzy *Maître de conférences (HDR), Université Clermont Auvergne*	Directeur de thèse
Ioan Todinca *Professeur, Université d'Orléans*	Rapporteur
Nicolas Trotignon *Directeur de recherche, CNRS, ENS de Lyon*	Président du jury

Remerciements

Mes premiers remerciements vont naturellement à Vincent Limouzy, ainsi qu'à Aurélie Lagoutte et Lucas Pastor, pour leur encadrement pendant cette thèse au LIMOS. Je pourrais faire une énumération de toutes les raisons pour lesquelles je vous remercie mais je ne vais pas le faire, je laisse l'énumération pour le contenu proprement dit de ce manuscrit. Je voudrais surtout vous remercier d'avoir cru en moi et de m'avoir fait confiance quand je suis partie en disant que je reviendrais.

Merci Ioan Todinca d'avoir accepté d'être rapporteur de ma thèse, et merci pour vos commentaires enthousiastes sur mon travail. Roberto Grossi, thank you for having accepted to review my thesis, and for your constructive comments about my manuscript. Je remercie également Fatiha Bendali, Marthe Bonamy, Nofar Carmeli, Christian Laforest et Nicolas Trotignon pour avoir accepté de compléter mon (nombreux) jury, et pour les discussions intéressantes qui ont eu lieu lors de la soutenance.

Comment pourrais-je ne pas remercier Arnaud Mary pour tous les échanges et toutes les (bonnes) idées qui ont émergé de nos discussions et sessions de travail ? Maintenant c'est chose faite, et j'espère qu'on aura d'autres occasions d'échanger autour de l'énumération.

During this thesis, I had the opportunity to meet several people from around the world and enjoyed discussing with them. First, I would like to thank Takeaki Uno who welcomed me twice in his lab in Tokyo. Then, a big thank you for the Italian team (Alessio, Andrea, Giulia, Davide, Alice and the others) and the Japanese team (Kunihiro, Kazuhiro, Takeaki, and the others) ; all the discussions we had during and around the WEPA conferences were precious !

Enfin, merci aux autres personnes avec qui j'ai pu travailler pendant cette thèse : Aline, Benjamin, Thomas, travailler avec vous, même à distance, fut une vraie réjouissance pour moi lors de cet épisode désagréable que fut le Covid (que je ne remercie pas, d'ailleurs). Merci aussi à tous et toutes les membres d'AlCoLoCo pour les sessions de travail au labo ou à la campagne, qui ont fait émerger de jolis résultats mais aussi et surtout de jolis problèmes !

Je tiens à remercier sincèrement le personnel administratif et technique du LIMOS pour m'avoir fourni de bonnes conditions de travail, ainsi que Dominique Torrisani dont l'aide m'a été précieuse.

Qu'aurait été mon séjour au LIMOS s'il n'y avait eu toutes les personnes avec qui j'ai partagé mon bureau (ou presque) : Oscar, Simon, Alexey, Alexis, Élodie, Trang, Aurélien, Loan, Mari et les autres. Merci à vous pour la bonne humeur et pour les pauses café. Merci également au cerisier du LIMOS qui agrémente tant ombrageusement que gustativement le début de l'été.

Mais pendant ces ~~trois~~ quatre ans, il n'y a pas eu que la thèse ! J'aimerais donc remercier chaleureusement toutes les personnes (nombreuses !) qui ont contribué à la richesse de ma vie musicale pendant toutes ces années : par ordre d'apparition, Agnès Lacornerie et l'ensemble de flûtes du conservatoire de Clermont, l'atelier de danse trad, Maude et les Phonies Polies, le Département de Musique Ancienne de l'ENM de Villeurbanne tout entier, l'Académie d'été de l'ensemble Artifices, Iels en Voix, la Chapotée, Yufang, Ophélie Berbain, et j'en oublie sûrement.

Un IMMENSE merci à Frédérique Thouvenot ; merci à toi de m'avoir reprise dans ta classe de flûte à bec à Villeurbanne, merci pour ton enseignement et ta passion pour la flûte à bec (merci aussi pour le shakuhachi !). Merci à vous Élisa, Julie, Alexis, Juliette, Célia, et merci aussi aux flûtes de consort (je ne les oublie pas ♡). Sans cette année passée loin de mon bureau à l'ISIMA, je crois que cette thèse aurait été beaucoup moins bien.

La MMI et ses membres ont contribué à la richesse de ma vie mathématique, alors que j'étais loin de ma thèse. Ce fut une belle expérience que j'espère réitérer si j'en ai l'occasion.

Merci également à toutes les personnes qui m'ont hébergée chez elles à l'occasion de conférences, contribuant elles aussi à la richesse de ma vie mathématique : François, Sabine, Antonin, Étienne…

Merci à tou·tes les ami·es auprès de qui j'ai pu me plaindre ou me réjouir pendant cette thèse ; Léo, Pedro, Mickaël, Alide, Antonin, tou·tes les ami·es du club impro / BDthêk et tou·tes celleux que j'oublie (pardonnez-moi). Mathilde, ton soutien est précieux et tes histoires parfois bien rigolotes. Marie-France, copine de danse et d'autres choses aussi, et surtout voisine perpétuelle, j'ai beaucoup aimé faire ta connaissance et traîner un peu au marché de Clermont ou à Aubière avec toi. Enfin, Alexis, merci pour la maison, pour la randonnée et pour les expériences culinaires et bricolagesques. Habiter avec toi, c'était bien.

Pour finir, merci à ma famille. Déménager presque tous les ans ne fut pas de tout repos et je vous m'y avez aidée à chaque fois !

Contents

Introduction (French)

Les graphes sont des structures combinatoires très simples. Ils peuvent être vus comme des représentations abstraites des interactions entre divers objets. En dépit de leur apparente simplicité, les graphes sont des outils puissants quand il s'agit de modéliser de nombreuses situations de la vie réelle. Leurs applications sont variées : recherche opérationnelle (allocation de ressources, ordonnancement), Internet mondial (liens entre les pages web), réseaux de communication, mais aussi réseaux routiers, transports et véhicules autonomes, entre autres. Ces structures combinatoires sont présentes dans de nombreux domaines scientifiques : on les rencontre par exemple en biologie (arbres phylogénétiques, génétique, interactions protéines-protéines), en chimie (structures des molécules), en sociologie (analyse des réseaux sociaux), en neurosciences (interactions entre les différentes zones du cerveau)...

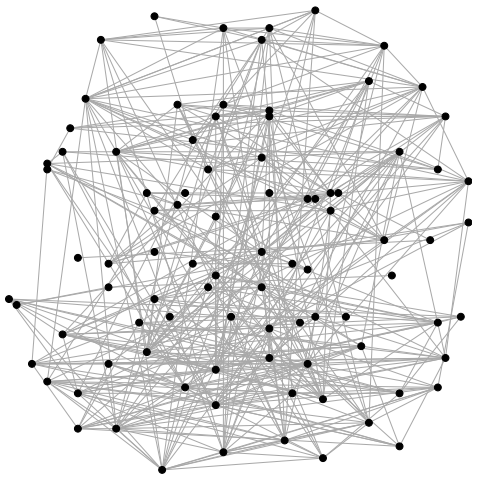

Figure 1 : Un graphe construit à partir de données d'IRMf au repos sur le cerveau d'un patient dans le coma. Données : Achard *et al.*, 2012 [1].

Pour illustrer une de ces applications, la Figure 1 représente un graphe construit à partir de données médicales sur le cerveau d'un patient dans le coma [1]. Il modélise

les 90 zones identifiées du cerveau et 400 connexions entre ces zones. Dans ce cas, la représentation des données sous forme de graphe est utilisée pour étudier comment l'organisation du cerveau change chez les patients dans le coma.

Peut-être parce que les graphes sont omniprésents en science, la théorie des graphes est un domaine de recherche très actif. Les problèmes étudiés sur les graphes concernent à la fois les aspects algorithmique et structurel. En théorie des graphes, les problèmes les plus généralement étudiés sont des *problèmes d'optimisation* : quel est le nombre minimal de couleurs dont on a besoin pour colorer un graphe donné ? Comment peut-on trouver le nombre minimum de nœuds à enlever pour qu'un graphe acquière une certaine propriété décidée à l'avance ? Peut-on calculer efficacement le plus court circuit passant par une liste de villes donnée avant de retourner à son point de départ ? En réalité, ces problèmes sont pour la plupart difficiles (NP-difficiles ou NP-complets), et sont de ce fait étudiés sous divers angles.

Parfois il est difficile de résoudre des problèmes d'optimisation, mais plus souvent encore, calculer seulement une solution d'un problème n'a en fait même pas de sens. Par exemple, lorsque l'on s'intéresse à une base de données, la requête la plus courante est la *liste* de toutes les entrées qui satisfont certaines conditions : la liste des personnes vivant dans une ville donnée, la liste des étudiants qui ont réussi un examen, la liste des itinéraires possibles pour atteindre une destination... Dans d'autres situations comme des applications en biologie ou en statistiques [27, 12], la notion de « meilleure » solution peut être très difficile à modéliser en termes de graphes. Il est plus intéressant dans ce cas de pouvoir générer un échantillon de « bonnes » solutions (pour une définition appropriée de « bonne »), et laisser l'utilisateur·rice choisir celle qui convient le mieux selon ses critères.

C'est pourquoi on s'intéresse aux *algorithmes d'énumération* qui, à l'inverse des algorithmes d'optimisation, renvoient toutes les solutions d'un problème (ou au moins toutes les « bonnes » solutions). En général (mais pas toujours), pour trouver toutes les solutions du problème, on commence par en trouver une et par des modifications sur les solutions déjà trouvées, on en génère de nouvelles. Pour faire cela, on doit s'assurer que les règles de modifications locales permettent vraiment de visiter *toutes* les solutions possibles.

L'approche énumérative présente quelques autres spécificicités. En particulier, le nombre de solutions qu'un algorithme d'énumération doit renvoyer peut être vraiment énorme comparé à la taille de l'entrée. Dans une telle situation, qu'entend-on par *efficace* pour un algorithme d'énumération ? Le temps d'exécution de l'algorithme est-il encore borné en fonction de la taille de l'entrée, ou bien doit-on aussi prendre en compte le nombre de solutions ? Dans ce cas précis, la notion d'efficacité a été adaptée pour prendre en compte les spécificités des algorithmes d'énumération [57].

En pratique, on cherche à générer toutes les solutions l'une après l'autre en garantissant un temps court entre deux solutions. Les algorithmes présentés dans cette

thèse ont ce qu'on appelle un *délai polynomial*, ce qui signifie que le temps passé à attendre entre deux solutions sera polynomial en la taille de l'entrée. Ces algorithmes sont considérés comme rapides en théorie.

Dans cette thèse, je m'intéresse aux problèmes de modification de graphes. Imaginons une situation dans laquelle des données ont été acquises et transformées en graphe, et la structure des données implique des propriétés structurelles spécifiques sur le graphe qui en résulte. Les propriétés structurelles sont importantes car elles permettent de faire des calculs beaucoup plus rapidement. Malheureusement, pendant l'envoi, certaines des données ont été altérées : le graphe a perdu sa propriété intéressante. Dans cette situation, il est naturel de vouloir essayer de réparer le graphe pour qu'il récupère sa propriété. Cependant, on ne sait pas comment était le graphe avant envoi.

Pour « réparer » le graphe, nous avons trois choix : enlever des nœuds, enlever des liens entre des nœuds ou au contraire ajouter des liens. Pour chaque problème considéré, on se cantonne à l'une de ces trois opérations. Le problème d'optimisation associé est celui de savoir quel est le nombre minimum de nœuds ou d'arêtes à enlever (ou ajouter) dans le but de donner au graphe la propriété voulue. Mais ce problème d'optimisation est bien souvent difficile [63] et c'est là que l'énumération devient utile.

Lorsqu'on essaye de résoudre notre problème d'énumération, on fait l'hypothèse que le nombre de données altérées n'est « pas trop grand ». Le principe est alors de générer un ensemble de réparations potentielles du graphe pour ensuite choisir la plus crédible, puisque la crédibilité d'une solution n'est pas toujours quelque chose qui se modélise facilement en termes mathématiques.

La question posée dans ce manuscrit est la suivante. Étant donnés un graphe G et une propriété de graphes $\mathcal{P}$, peut-on énumérer efficacement toutes les « réparations » de G qui possèdent la propriété $\mathcal{P}$? Il y a plusieurs manières d'attaquer le problème.

Une première approche consiste à fixer la propriété $\mathcal{P}$ en avance et à profiter des caractérisations connues de $\mathcal{P}$ pour énumérer toutes les « réparations » de G qui satisfont $\mathcal{P}$. Cette approche peut être très puissante et donner lieu à des algorithmes très efficaces comme ceux présentés dans le Chapitre 3. Cependant, les algorithmes obtenus ainsi sont en général difficiles à adapter à d'autres propriétés de graphes.

L'autre approche consiste à chercher un algorithme général qui, étant donnés un graphe G *et* une propriété $\mathcal{P}$ (et éventuellement une opération sur les graphes spécifique à $\mathcal{P}$), renvoie toutes les « réparations » de G possédant la propriété $\mathcal{P}$, sans fixer $\mathcal{P}$ en avance. Des progrès récents ont été faits sur l'approche générale [22, 23]. Dans ce but, j'étudie et améliore dans ma thèse une technique générale existante appelée *Proximity Search*.

Je m'intéresse ici à quatre classes de graphes bien connues : graphes *split*, cographes, graphes *threshold* et graphes cordaux. Ces classes de graphes sont reliées à des paramètres de graphes très importants pour de nombreuses applications.

Le reste du manuscrit est rédigé en anglais. J'en donne ici un résumé succinct en français.

Premièrement, le Chapitre 1 contient quelques éléments de contexte sur les graphes en général, les définitions des objets et concepts utilisés dans la suite, sans prétendre à l'exhaustivité. Les *complétions et délétions minimales* d'un graphe dans une classe donnée sont définies formellement. Ces objets seront au centre de notre attention, ainsi que les sous-graphes maximaux induits.

Ensuite, le Chapitre 2 est consacré à l'*énumération* et ses aspects algorithmiques. Avant de relier formellement l'énumération avec les complétions et délétions minimales de graphes, je présente des techniques d'énumération connues pour donner de bons résultats en termes d'efficacité.

Une classe de graphes particulière, les graphes *split*, est étudiée dans le Chapitre 3. Deux algorithmes efficaces sont proposés : un pour l'énumération des complétions minimales en graphe *split* et un pour celle des sous-graphes *split* induits maximaux. Pour les complétions *split* minimales, une bijection est établie avec les stables maximaux d'un graphe auxiliaire. En utilisant des résultats connus sur l'énumération des stables maximaux [84], je montre que l'énumération des complétions *split* minimales est possible avec délai polynomial en espace polynomial. Quant au cas des sous-graphes *split* induits maximaux, la technique du *problème restreint* introduite par Cohen, Kimelfeld & Sagiv en 2008 [22] est appliquée pour trouver un algorithme à délai polynomial en espace polynomial également. Les algorithmes obtenus pour les graphes *split* sont très efficaces mais ils sont aussi très spécifiques et ne peuvent être facilement adaptés pour d'autres classes de graphes.

Pour trouver une technique générale efficace, on s'intéresse au célèbre *Flashlight Search* dans le Chapitre 4. C'est une technique conceptuellement simple qui peut se révéler très efficace dans certains cas. L'énumération d'objets par *Flashlight Search* passe par la résolution répétée d'un sous-problème appelé le *problème d'extension*. Je prouve que ce problème est difficile (NP-difficile ou NP-complet) si aucune hypothèse n'est faite sur la classe de graphes considérée, ce qui empêche la technique d'être aussi générale que nous le souhaiterions.

C'est pourquoi dans le Chapitre 5, on s'intéresse au *Proximity Search*, une technique récemment formalisée par Conte & Uno [26], qui semble en un sens être plus générale que d'autres techniques d'énumération connues et permet d'atteindre de bons résultats de complexité. J'applique cette technique pour résoudre plusieurs problèmes, notamment à l'énumération des sous-cographes induits et des sous-graphes *threshold* induits d'un graphe. Enfin, le *Proximity Search* est utilisé pour énumérer les complétions minimales en graphe cordal avec délai polynomial, une question qui a suscité beaucoup d'intérêt dans les dernières années [18].

Cette approche est généralisée pour étendre la technique de *Proximity Search*

utilisée en pratique. Deux extensions – dans différentes directions – en sont exposées au Chapitre 6. L'une permet d'élargir l'ensemble des problèmes auxquels on peut appliquer le *Proximity Search*, l'autre de réduire la complexité en espace dans certains cas.

En parallèle des algorithmes d'énumération, j'ai aussi participé à d'autres projets de recherche concernant la théorie des graphes. Les résultats obtenus se rapportent à des aspects plus structurels des graphes.

L'un de ces projets concerne les *colorations gloutonnes connexes* de graphes. La question que l'on se pose est la suivante : quels sont les graphes pour lesquels aucun algorithme de coloration gloutonne connexe n'utilise le nombre optimal de couleurs ? L'autre projet a pour sujet les *ensembles dominants localisateurs*. Étant donné un graphe orienté D, quelle est la taille minimum d'un ensemble de sommets qui soit à la fois dominant (chaque sommet est soit dans l'ensemble, soit a un voisin dans l'ensemble) et localisateur (deux sommets peuvent être différenciés par leurs voisinages dans l'ensemble) ? Dans chaque situation, en se restreignant à un graphe d'entrée qui satisfait certaines propriétés, nous avons partiellement répondu à ces questions.

Les deux problèmes mentionnés ici ne sont pas présentés dans la thèse elle-même, mais un résumé de mes contributions (en anglais) se trouve à la fin du manuscrit.

Introduction

Graphs are very simple combinatorial structures. They can be seen as abstract representations of objects and their interactions. Despite their apparent simplicity, graphs are powerful tools when it comes to modelling many real-life situations. Their applications are various: operations research (resources allocation, scheduling), the world wide web (links between webpages), communication networks, but also roads, transportation, and autonomous vehicles, among others. These combinatorial structures are found in many – if not all – scientific fields: they arise for example in biology (phylogenetic trees, genetics, protein-protein interactions), chemistry (structures of molecules), sociology (social network analysis), neurosciences (interactions between the different zones of the brain)...

As an illustration, Figure 1 represents a graph obtained from medical data on the brain of a comatose patient [1]. It models the 90 identified zones of the brain and 400 connections between them. In this case, the representation of data in the form of a graph is used to study how the organisation of the brain changes for a person in a coma.

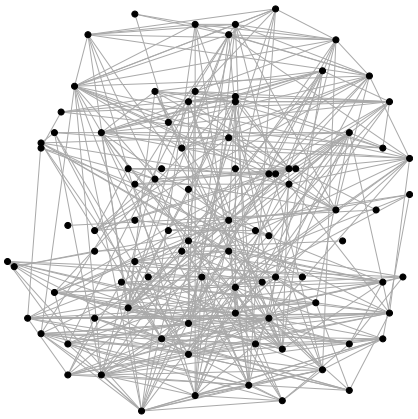

Figure 1: A graph obtained from fMRI data on the brain of a comatose patient. Data: Achard *et al.*, 2012 [1].

Perhaps due to the omnipresence of graphs in science, graph theory – the study of graphs – is nowadays a very active branch of research. The problems studied on graphs concern both the structural and the algorithmic aspects. In graph theory, the problems typically studied are *optimisation problems*: what is the minimum number of colours needed to colour a given graph? How can we find the minimum number of nodes one has to remove to acquire some desired property? Can we compute efficiently the shortest circuit going through a prescribed set of cities before going back home? In fact, these questions are usually hard (NP-hard or NP-complete), and are therefore studied from many angles.

Sometimes, solving optimisation problems is hard, yet even more often it is just not meaningful to generate only one solution of a problem. For example, when dealing with databases, the typical request is the *list* of all entries satisfying some specified conditions: list of people living in a given city, list of students who have passed an exam, list of possible itineraries to reach a destination...

In some other situations such as applications in bioinformatics or statistics [27, 12], the notion of "best" solution can be hard to model in terms of graphs. It is more interesting in this case to be able to generate a sample of "good" solutions (for an appropriate definition of "good"), and let the user choose themself the best suited solution for their purpose.

This is why we are interested in *enumeration algorithms* which, unlike optimisation algorithms, return all the solutions of a given problem (or at least all the "good" ones). In general (but not always), to find all the solutions of a problem we begin with finding one, and then slightly modify the solutions already found to discover new ones. Doing this, we need to ensure the local modification rules really allow us to visit *all* the possible solutions.

The enumeration approach has other specific issues. In particular, the number of solutions that have to be computed by an enumeration algorithm can be really huge compared to the input size. In such a situation, what does it mean for an enumeration algorithm to be *efficient*? Is the running time of the algorithm still bounded in terms of the input size, or is it necessary to take the number of solutions into account? The notion of efficiency for enumeration algorithms has therefore been adapted to take these issues in consideration [57] (see Chapter 2 for more details).

In practice, we seek to generate all the solutions one after the other while guaranteeing the shortest possible time between two solutions. The algorithms mentioned in the present manuscript run with *polynomial delay*, meaning the time spent between two solutions is polynomial in the input size. These algorithms are considered fast in theory.

In this thesis, we are interested in graph modification problems. Imagine a situation where some data have been acquired and transformed into a graph, and the structure

of the data implies some specific structural property on the graph. The structural property is important because it permits to run the computations faster. Unfortunately, during the transfer or gathering, some of the data have been altered: the graph lost its interesting property. In this situation, a natural thing to do is try to repair the graph so that it gets the interesting property again. However, we do not know how the graph was before it was sent.

To "repair" the graph, we have three choices: removing some nodes, removing some links between nodes, or on the contrary adding some links. For each problem we consider, we choose one of these operations and only make this sort of changes. The associated optimisation problem asks for the minimum number of elements to remove (or add) in order to make the graph acquire the desired property. However, such an optimisation problem is usually hard [63], and this is where enumeration becomes useful.

When trying to solve our problem with enumeration, we assume that the amount of alterations is "not too large". Then, the principle is to generate a bunch of potential reparations of the graph, and choose the most credible one, since the credibility of a solution is not always something easy to model mathematically.

The question we address in the present thesis is the following. Given a graph G and a graph property $\mathcal{P}$, can we enumerate efficiently all "reparations" of G into the property $\mathcal{P}$? There are several ways to approach the question.

First, one can fix the property $\mathcal{P}$ in advance and benefit from the known characterisations of $\mathcal{P}$ to enumerate all "reparations" of G in $\mathcal{P}$. This can be very powerful and provide efficient algorithms, as those designed in Chapter 3. However, algorithms obtained this way cannot be adapted easily to other properties.

The other approach consists in looking for a general algorithm which, given a graph G *and* a property $\mathcal{P}$ (plus maybe some operation specific to $\mathcal{P}$), returns all the "reparations" of G into the property $\mathcal{P}$, without fixing $\mathcal{P}$ in advance. Recent progress has been made on the general approach [22, 23]. In Chapter 6 of the present thesis, we also study and improve an existing general method for this purpose.

We are interested here in four well known graph properties that are related to interesting parameters in graphs. These properties are detailed in Chapter 1.

The manuscript is organised as follows.

First, in Chapter 1, we give some necessary background about graphs: definitions of the basic objects and concepts used in the sequel, and we introduce *minimal completions and deletions* of a graph into a prescribed class. These objects will be at the centre of our attention, together with maximal induced subgraphs.

Then, Chapter 2 is devoted to the introduction of *enumeration* and its algorithmic

aspects. Before linking formally enumeration with minimal graph completions and deletions, we present some of the most famous existing techniques that imply efficient enumeration algorithms under certain conditions.

We will have a closer look on a particular class of graphs, *split graphs*, in Chapter 3. Two problems are considered on split graphs; both of them give different insights on subgraph enumeration problems. However, the algorithms we obtained for split graphs are very specific.

In order to look for a general technique, we investigate the famous *Flashlight Search* in Chapter 4. We prove the difficulty of a sub-problem, the *extension problem*, preventing the technique to be as general as we would like it to be.

This is why in Chapter 5, we are interested in a recently introduced enumeration technique, *Proximity Search*, which seems to be in some sense more general than the previously known ones and achieves good time complexity. This technique is here successfully applied to several problems: enumeration of maximal sub-cographs and threshold subgraphs, and the challenging question of minimal chordal completions enumeration.

Generalising our approach leads us to extend the usual Proximity Search-based technique used in practice. Two extensions of it – in different directions – are exposed in Chapter 6, allowing us to enlarge the set of problems that can be solved, and to reduce the space usage in some cases.

In parallel to enumeration algorithms, other scientific projects about graphs have been joined during this thesis. The results we obtained refer to more structural aspects of graphs.

One of the projects is about *connected greedy colourings* of graphs. The question we addressed is the following: what are the graphs for which no connected greedy colouring algorithm uses the optimal number of colours? The other project was about *locating-dominating sets*. Given a directed graph D, what is the minimum size of a set of nodes that is both dominating (every node is linked to a node in the set) and locating (two different nodes have different neighbours in the set)? By restricting each time the graph to satisfy a specific property, we were able to partially answer those questions.

The two aforementioned problems will not be addressed in the present document, but a summary of these contributions can be found at the end of the manuscript.

Chapter 1

Graphs, completions and deletions

This chapter contains a brief introduction to the basic objects in which we are interested: graphs. More information on this topic can be found in several works including the reference books written by Diestel [36] and Golumbic [49]. After giving some general information about graphs, we will detail the problems that are studied in all the remaining of this thesis.

ALL THE PROBLEMS mentioned in this work are problems on *graphs*. This is why in this chapter, we will give the necessary background and notations on graph theory that will be useful in the sequel. After that, we describe more formally the problems in which we are interested.

1.1 Graphs and properties

1.1.1 General definitions and notations

This section is intended to present in a unified way the notations that will be used in the rest of the thesis. Standard graph theory notations are adopted, such as those provided in Diestel's book [36].

In the present work, graphs as "basic objects" – those on which the problems are defined – are always considered simple and undirected. Nevertheless, on a higher level, we will also be led to consider some oriented graphs.

Definition 1.1 (Graph)**.** *A* graph *is a couple of sets $G = (V, E)$ such that all elements of E are two-element-subsets of V. The elements of V are called the* vertices *of G, those of E are called its* edges.

For the remainder of this section, let $G = (V, E)$ be a graph. When the context is ambiguous, for example when several graphs are involved, we may use the notations $V(G)$ and $E(G)$ to precise which graph we are considering.

Edges and neighbourhoods. For an edge $\{u, v\} \in E$, we will often use the simpler notation uv. The vertices u and v are called *adjacent* if $uv \in E$, and the edge uv is said to be *incident* to its endpoints u and v.

The *neighbourhood* of a vertex v in G is denoted by $N_G(v)$ and defined as the set of vertices adjacent to v. These vertices are called the *neighbours* of v. More formally, $N_G(v) := \{w \in V(G) \mid vw \in E(G)\}$. The number of neighbours of a vertex v is called its *degree*, denoted by $d_G(v)$. The *closed neighbourhood* of a vertex v in G is defined as $N_G[v] := N_G(v) \cup \{v\}$. When it is clear from the context, the subscript G might be omitted.

Whenever two vertices x and y are such that $N(x) \setminus \{y\} = N(y) \setminus \{x\}$, they are called *twins*. If x and y are twins and $xy \in E$, that is, if $N[x] = N[y]$, then x and y are called *true twins*. Otherwise, $N(x) = N(y)$ and x and y are called *false twins*. These two situations are illustrated in Figure 1.1.

false twins true twins

Figure 1.1: The two vertices x and y have the same neighbours: they are (true or false) twins.

A *path* in G between two vertices u and v, also called a u-v-path, is a sequence of distinct vertices $v_0 = u, v_1, \ldots, v_k = v$ such that for any $1 \leqslant i \leqslant k$, $v_{i-1}v_i \in E$. The graph G is *connected* if for any two vertices u and $v \in V$, there exists a u-v-path, that is, a path between u and v, in G.

A *cycle* is a sequence of distinct vertices $v_0, v_1, \ldots, v_k = v_0$ such that for any $1 \leqslant i \leqslant k$, $v_{i-1}v_i \in E$. A graph without any cycle is called a *tree*.

The *complement* of a graph G is the graph $\overline{G}$ on the vertex set V whose edge set is exactly the set of two-element-subsets of V that are not in E. More formally, $\overline{G} = (V, \{\{u, v\} \mid u \neq v \in V \text{ and } \{u, v\} \notin E\})$. Therefore, two adjacent vertices in G are non-adjacent in $\overline{G}$, and conversely two non-adjacent vertices in G are adjacent in $\overline{G}$.

Finally, given X, Y two subsets of vertices of G, the *symmetric difference* between X and Y, denoted by $X \triangle Y$, is defined as $(X \cup Y) \setminus (X \cap Y)$.

Subgraphs and supergraphs. Let $X \subseteq V$ be a subset of vertices of G. We denote by $G[X]$ the subgraph of G *induced* by X, defined as $G[X] = (X, E(G) \cap \binom{X}{2})$, where the notation $\binom{X}{2}$ stands for the set of all two-element subsets of X. Alternatively, $G[X]$ is obtained from G by removing all the vertices which do not belong to X, and all the edges incident to at least one deleted vertex. For simpler notations, for a set $Y \subseteq V$ we might sometimes write $G \setminus Y$ instead of $G[V \setminus Y]$.

A graph H is called an *induced subgraph* of G if there exists $X \subseteq V$ such that $G[X]$ is isomorphic to H[1].

For a graph H, if G does not contain a copy of H as an induced subgraph, the graph G is called *H-free*.

A graph H is a *subgraph* of G if H can be obtained from G by removing some vertices and some edges of G, including those which were incident to at least one deleted vertex, that is to say if $V(H) \subseteq V(G)$ and $E(H) \subseteq (E(G) \cap \binom{V(H)}{2})$. Of course, induced subgraphs are particular subgraphs where the last inclusion is an equality, *i.e.* $E(H) = (E(G) \cap \binom{V(H)}{2})$. If in addition $V(H) = V(G)$, then H is called an *edge-subgraph* of G.

A graph H is called an *edge-supergraph* of G, if $V(H) = V(G)$ and $E(G) \subseteq E(H)$. Remark that H is an edge-supergraph of G if and only if its complement $\overline{H}$ is an edge-subgraph of $\overline{G}$.

Graph properties. Since we are interested in graphs that satisfy a certain graph property, let us have a look on the different kinds of graph properties.

By abuse of language, we will often identify a graph property with the set of graphs satisfying it, and write "the *class* $\mathcal{P}$" to denote the set of all graphs that fulfil the property $\mathcal{P}$.

Let us denote by $\overline{\mathcal{P}} = \{\overline{G} \mid G \in \mathcal{P}\}$ the complementary class of $\mathcal{P}$, that is, the class of all graphs whose complement fulfils the property $\mathcal{P}$. Then, if $\mathcal{P} = \overline{\mathcal{P}}$, we will say that the class $\mathcal{P}$ is *self-complementary*.

A graph property $\mathcal{P}$ is called *hereditary* when it is closed under vertex removal. In other words, if the property is fulfilled by the graph we consider, then it is also fulfilled by all its induced subgraphs. A property $\mathcal{P}$ is said to be *connected-hereditary* if, when a graph satisfies $\mathcal{P}$, all its *connected* induced subgraphs also satisfy $\mathcal{P}$. All the properties considered in this thesis are hereditary. Typically, being H-free (for a given graph H) is a hereditary property. In this case, H is called a *forbidden induced subgraph* for the property. When a graph property $\mathcal{P}$ is defined by a family $\mathcal{F}$ of forbidden induced subgraphs – equivalently, $\mathcal{P}$ corresponds to being $\mathcal{F}$-free, or H-free for any $H \in \mathcal{F}$ –, the elements of $\mathcal{F}$ are called *obstructions* for $\mathcal{P}$.

[1]That is to say, there must exist a bijection $\varphi : X \longrightarrow V(H)$ such that for any two vertices u and v in X, $uv \in E(G)$ if and only if $\varphi(u)\varphi(v) \in E(H)$.

A graph property is called *monotone* when it is closed under both edge removal and vertex removal. That is to say, if the property is satisfied by G, then it is also satisfied by all subgraphs of G.

A property $\mathcal{P}$ is called *sandwich-monotone* [56] if, for any two graphs G and H on the same vertex set V, that both fulfil the property $\mathcal{P}$, and such that H is an edge-subgraph of G, there exists a sequence of edges $e_1, e_2, \ldots, e_k$ of $E(G) \setminus E(H)$ such that for any $1 \leqslant i \leqslant k$, the graph $(V, E(H) \cup S_i)$ verifies property $\mathcal{P}$, where $S_i = \{e_1, \ldots, e_i\}$. In other words, there exists a sequence of single-edge removals that allows us to go from G to H. In the general framework of set systems, being sandwich-monotone is equivalent to being a *strongly accessible* set system [24].

Directed graphs. Some graphs that will be considered here are *directed graphs*, meaning that their edges have a direction. They will not be encountered as the "basic objects" on which the problems are defined, but rather as "meta-structures" in which the directed edges represent ways to transform an object into another.

Definition 1.2 (Directed graph)**.** *A* directed graph *is a couple of sets $D = (V, A)$ such that all elements of A are couples of elements of V, that is, $A \subseteq V \times V$. The elements of A are called the* arcs *of D and the elements of V are its* vertices*.*

With this definition, the neighbours of a vertex u are split into two categories: these for which $(u, v) \in A$, called its *out-neighbours*, and those such that $(v, u) \in A$, called its *in-neighbours*.

If there exists an oriented path – a sequence of vertices $v_0, v_1, \ldots, v_k$ such that for any $1 \leqslant i \leqslant k$, $(v_{i-1}, v_i) \in A$ – between any two vertices of D, then D is called *strongly connected*.

Now that all the necessary notions are defined, let us have a closer look on the objects that will be of interest in the remaining of the present manuscript.

1.1.2 Some remarkable graph classes

In this thesis, we are particularly interested in some well-known graph classes, namely *chordal graphs*, *split graphs*, and *cographs*. These have been well studied and characterised in the literature. In this section, we give the definitions of those graph classes.

Chordal graphs. *Chordal graphs*, also called *triangulated graphs* or sometimes *rigid circuit graphs* have been studied since the 1960s [37]. They constitute a well-studied class of graphs since they arise in many practical applications [18, 27, 49, 69].

Definition 1.3 (Chordal graphs)**.** *A graph is* chordal *if it does not contain any induced cycle of length 4 or more.*

In other words, every cycle of length more than 3 of a chordal graph has at least one chord (*i.e.* an edge that connects non-consecutive vertices of the cycle). Equivalently, any induced cycle of a chordal graph has length 3. These are simply reformulations of the definition of chordal graphs.

Note that, since removing vertices cannot create long induced cycles, chordal graphs form a hereditary graph class.

A chordal graph is represented in Figure 1.2. The obstructions of chordal graphs are the long induced cycles, as shown on the right.

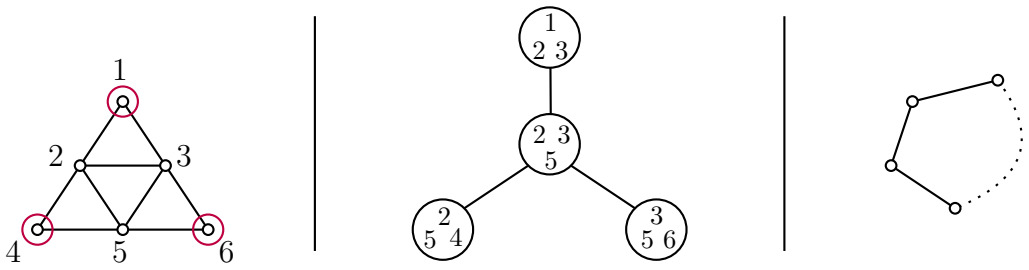

Figure 1.2: A chordal graph with its simplicial vertices identified. A corresponding clique tree is shown in the middle. The obstructions are long cycles (on the right).

Chordal graphs benefit from characterisations of various types. For example, and perhaps most importantly, it has been shown by Dirac in 1961 that any chordal graph has at least two *simplicial* vertices, that is, vertices whose neighbourhoods induce a clique [37]. The simplicial vertices are identified on the graph of Figure 1.2. Since the class of chordal graphs is hereditary, this implies that simplicial vertices can be recursively removed from a chordal graph until there is no vertex left. This gives an ordering on the vertices of the graph, called a *perfect elimination ordering*. In fact, chordal graphs are exactly the graphs in which such an ordering exists. Because of the existence of a perfect elimination ordering, it has been shown that chordal graphs can be recognised in linear time [82].

Several other characterisations are known for chordal graphs. To cite a few of them, chordal graphs can be seen as intersection graphs of paths in a tree [48], or characterised in terms of minimal clique separators [37].

Chordal graphs possess some very interesting properties. Indeed, several problems that are usually hard on general graphs (*e.g.* graph colouring) become polynomial on chordal graphs [49]. On top of that, chordal graphs are closely connected to an important graph parameter called *treewidth* and its associated *tree-decomposition* introduced independently by Halin [52] and Robertson & Seymour [73].

A *tree-decomposition* of a graph G is a tree whose vertices are sets of vertices of G, called the *bags* of the decomposition. Moreover, it is required that each vertex appears

in at least one bag, for each edge there exists a bag containing its two endpoints, and for any vertex v of G, the subgraph induced by the bags containing v is a tree. Chordal graphs have been characterised as the graphs for which there exists a tree-decomposition whose bags all induce cliques in the original graph [48, 73]. For chordal graphs, such a decomposition is called *clique tree*. The clique tree associated with the graph of Figure 1.2 is shown in the middle.

Split graphs. The class of *split graphs* was introduced in the 1970s by Foldes & Hammer [42]. It is a well-studied subclass of chordal graphs [43, 49, 53, 65]. Its definition is the following.

Definition 1.4 (Split graphs)**.** *A graph G is* split *if its vertex set can be partitioned in $K + S$ where K is a clique of G and S is a stable set of G. The partition $K + S$ is called a* split partition *of G.*

As a direct consequence of the definition, the class of split graphs is self-complementary, meaning that for any split graph G, its complement $\overline{G}$ is also split. Moreover, split graphs form a hereditary graph class.

Note that the split partition of a split graph is not necessarily unique. Indeed, a vertex belonging to the clique but with no neighbour in the stable set could as well be placed in the stable set. Conversely, a vertex in the stable set that is adjacent to every vertex of the clique can be transferred to the clique without any consequence. Such a situation is illustrated in Figure 1.3, where the coloured vertex can be placed in any set of the split partition.

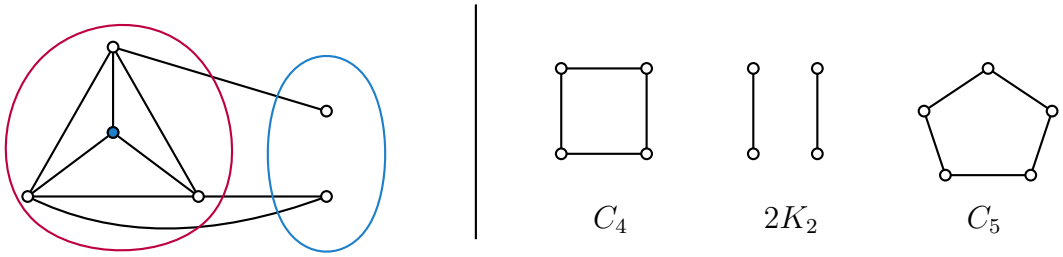

Figure 1.3: A split graph with its split partition. The coloured vertex can be placed either in the clique or in the stable set. The obstructions are shown on the right.

The class of split graphs benefits from various characterisations [42]. Among other ones, split graphs correspond exactly to chordal graphs whose complement is also chordal. Moreover, they have been characterised by forbidden induced subgraphs, namely as $(C_4, 2K_2, C_5)$-free graphs. These obstructions are shown on the right of Figure 1.3. Finally, split graphs can be recognised in polynomial time by looking only at the degrees of their vertices [53].

Cographs. The class of *cographs* was introduced in the 1970s by several authors under various names such as *hereditary Dacey graphs* or D^**-graphs* [59, 81]; since then, this class has been widely studied and characterised [28, 29, 33, 51]. In particular, Corneil, Lerchs & Stewart [28] provided some very useful characterisations of cographs. One of the many equivalent definitions of cographs is the following.

Definition 1.5 (Cographs)**.** *The class of* cographs *is defined recursively:*

- *a single-node graph is a cograph;*
- *if G_1 and G_2 are cographs, then their disjoint union $G_1 \cup G_2$ is also a cograph;*
- *if G is a cograph, then its complement $\overline{G}$ is also a cograph.*

First, remark that the class of cographs is hereditary. By definition, this class is also self-complementary.

Cographs have been characterised as the graphs that do not contain any P_4 – that is, a path on four vertices – as an induced subgraph. Moreover, graphs in this class benefit from a unique tree representation, the *cotree* [28]. The *cotree* associated with a cograph is a rooted tree whose leaves are exactly the vertices of the cograph, and whose internal nodes are labelled either 0 or 1. Then, two vertices are connected in the cograph if and only if the least common ancestor[2] of the corresponding leaves in the cotree is labelled 1. Thanks to this representation, it has been proved that cographs can be recognised in linear time [29]. A cograph with its associated cotree is illustrated in Figure 1.4, where two pairs of twin vertices are identified.

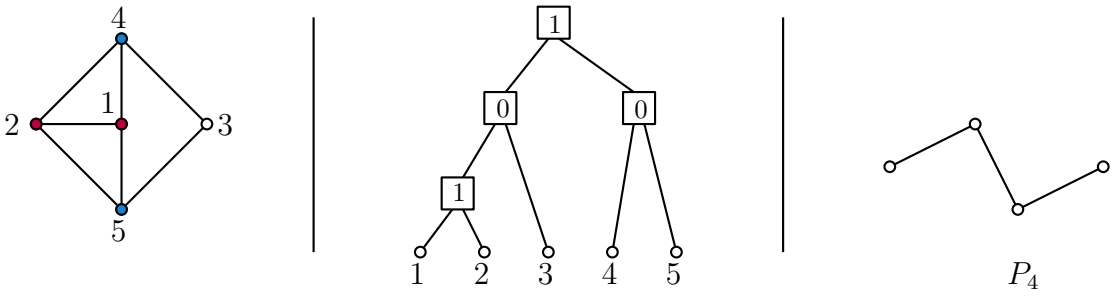

Figure 1.4: A cograph (on the left) with its associated cotree. In this graph, the vertices 1 and 2 are true twins, the vertices 4 and 5 are false twins. The obstruction P_4 can be seen on the right.

It has also been proved that in a cograph, there always exist two (true or false) twin vertices [28]. Even more, all cographs can be constructed by beginning with a single vertex and recursively adding a true or false twin to an already existing vertex. This provides what we call a *twin construction ordering* on the vertices of a cograph.

[2] the first common vertex encountered when going from the leaves to the root

We call *induced sub-cograph* of G an induced subgraph that is also a cograph; it is moreover maximal if it is inclusion-wise maximal among all induced sub-cographs of G.

Threshold graphs. *Threshold graphs* were introduced in 1977 by Chvátal & Hammer [21] as the set of graphs such that there exists a real number b and a weight function $w : V \mapsto \mathbb{R}$ such that for any subset X of vertices, X is a stable set if and only if $\sum_{v \in X} w(v) \leqslant b$. This rather complicated definition was motivated by applications to integer programming, and gave its name to the corresponding class of graphs: the number b can be seen as a threshold value for the existence of edges.

In the same paper [21, Theorem 1], this class of graphs has been characterised by a recursive construction involving *isolated* vertices – vertices that have no neighbour in the graph – and *universal* vertices, whose closed neighbourhood is the entire graph. We give here this characterisation as a simpler definition for the class of threshold graphs.

Definition 1.6 (Threshold graphs)**.** *A graph is* threshold *if it can be constructed from the empty graph by recursively adding isolated or universal vertices.*

Threshold graphs are at the intersection between split graphs and cographs. Therefore, they are characterised by forbidden induced subgraphs as the class of $(P_4, C_4, 2K_2)$-free graphs. A threshold graph is can be seen in Figure 1.5, and the obstructions are also represented. The class of threshold graphs is quite common in the literature; see for example the chapter dedicated to it in Golumbic's book [49].

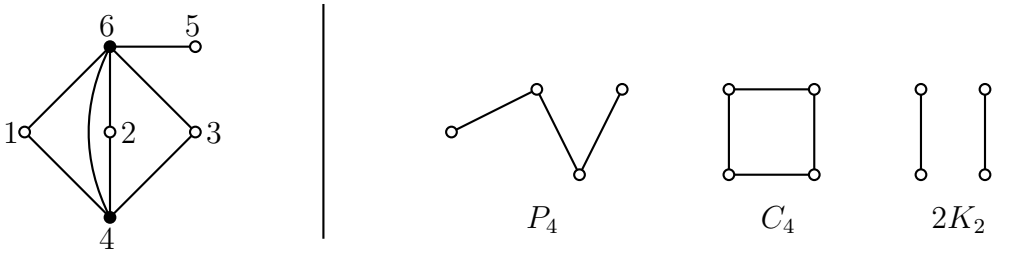

Figure 1.5: A threshold graph whose threshold construction ordering is 123456. White vertices were added as isolated, black vertices were added as universal. The three obstructions are presented on the right.

Like the split and cograph properties, the threshold property is hereditary and closed under complement. Moreover, like split graphs, threshold graphs can be recognised in linear time through their degree sequences [53]. Finally, note that the construction of a threshold graph by adding isolated and universal vertices naturally implies an ordering on its vertex set, that is called the *threshold construction ordering*.

A recap of the inclusions between the four graph classes presented in this section is given in Figure 1.6. A graph class is above another if it is larger, and the arcs are directed towards the smallest class. For the sake of completeness, the class of *Trivially Perfect graphs*, corresponding to the intersection between cographs and chordal graphs, is also represented on the picture. All the graph classes considered here are hereditary, and the coloured ones are self-complementary in addition.

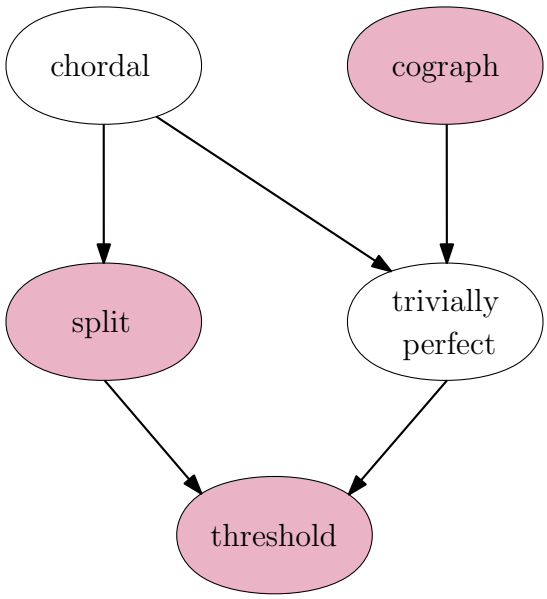

Figure 1.6: Diagram showing the inclusions between all the mentioned graph classes. An arrow shows an inclusion (directed towards the smallest class).

1.2 Minimal completions and deletions

Given an arbitrary graph G, there is no reason for it to fulfil a given property $\mathcal{P}$. Then, one of the questions we address is about computing induced subgraphs of G which satisfy $\mathcal{P}$. When the property is hereditary, finding such an induced subgraph is very easy: for example, the single-node graph is a trivial solution. However, finding such trivial solutions is not really satisfactory: we want to preserve somehow the structure of G, and find an induced subgraph that satisfies $\mathcal{P}$ and is "as close as possible" to G. In other words, we want to remove as few vertices as possible from G in order to make the graph fulfil property $\mathcal{P}$.

Unfortunately, finding the minimum number of vertices that have to be removed from G is in general a hard problem. This problem, known as the *node-deletion problem* has been shown to be NP-complete for most hereditary properties [63].

The other problem in which we are interested here is that of computing edge-subgraphs (resp. supergraphs) of G that satisfy $\mathcal{P}$ and are "close" to G. Like the previous one, this problem corresponds to removing (resp. adding) as few edges as possible from G to make the graph satisfy $\mathcal{P}$. Less general complexity results are known for the computation of the minimum number of edges that must be removed from (resp. added to) G in order to obtain a graph fulfilling $\mathcal{P}$, but some have been proved in special cases. In fact, there are several properties for which this problem is also NP-complete [86, 50], for example when the property under consideration is being chordal, as studied in Chapters 5 and 6.

Since the problem of finding the minimum number of elements (vertices or edges) that must be removed to make a graph satisfy a given property is hard in general, let us focus on the easier task of finding such a set of elements that is *minimal for inclusion*. If such a set is of minimum cardinality, it is necessarily also minimal for inclusion. This is why all over this thesis, *minimal* (resp. *maximal*) will always refer to inclusion-wise minimal (resp. maximal), and never to minimum (resp. maximum) cardinality.

We introduce here the objects studied in all the sequel: minimal $\mathcal{P}$-completions and deletions, and maximal induced $\mathcal{P}$-subgraphs of a graph $G = (V, E)$.

Let $\mathcal{P}$ be a graph property. An induced subgraph $G[X]$ of G, on the vertex set X, that verifies $\mathcal{P}$ is called an *induced $\mathcal{P}$-subgraph* of G. It is *maximal* if for any subset of vertices $Y \subseteq V$, if $X \subsetneq Y$ then $G[Y]$ does not fulfil the property $\mathcal{P}$. By abuse of language, we will often identify an induced $\mathcal{P}$-subgraph $G[X]$ with its set of vertices X.

A graph H is a *$\mathcal{P}$-edge-subgraph* of G, also called a *$\mathcal{P}$-deletion* of G, if it is an edge-subgraph of G and it fulfils property $\mathcal{P}$. It is called a *maximal* $\mathcal{P}$-edge-subgraph, or a *minimal* $\mathcal{P}$-deletion, if it is inclusion-wise maximal, that is to say there is no $\mathcal{P}$-edge-subgraph H' of G such that $E(H) \subsetneq E(H') \subseteq E(G)$. If E is a subset of edges of G, we might by abuse of language call E' a deletion of G, referring to the graph $(V(G), E')$, thus identifying a deletion of G with its edge set.

Similarly, a graph H is a *$\mathcal{P}$-edge-supergraph* of G, also called a *$\mathcal{P}$-completion* of G, if it is an edge-supergraph and G and it fulfils property $\mathcal{P}$. It is called a *minimal* $\mathcal{P}$-edge-supergraph of G, or a *minimal* $\mathcal{P}$-completion, if it is inclusion-wise minimal: there is no $\mathcal{P}$-supergraph H' of G such that $E(G) \subseteq E(H') \subsetneq E(H)$. The edges uv belonging to H but not to G, that is, $uv \in E(H) \setminus E(G)$, are called the *fill edges* of the completion. Similarly as for deletions, completions will often be identified with the set of fill edges they induce.

When $\mathcal{P}$ refers to being in some known class of graphs, say being a split graph, we will omit the hyphen and simply say: a split edge-subgraph, a split deletion, a minimal split completion, and so on.

In general, when $\mathcal{P}$ is satisfied by all stable sets – as it is the case for the property of being chordal, split, cograph, or threshold –, we are ensured that a $\mathcal{P}$-deletion exists

for any input graph. It makes sense in this case to look for an algorithm computing the minimal $\mathcal{P}$-deletions of G without any restriction on G. Conversely, if $\mathcal{P}$ is verified by all cliques, then a $\mathcal{P}$-completion exists for any graph G, in which case we look for an algorithm computing the minimal $\mathcal{P}$-completions of G without any restriction on G.

Let us illustrate these notions in the particular case when $\mathcal{P}$ denotes the class of split graphs. In Figure 1.7, a graph is represented with a minimal split completion, a (non-minimal) split deletion, and a maximal induced split subgraph.

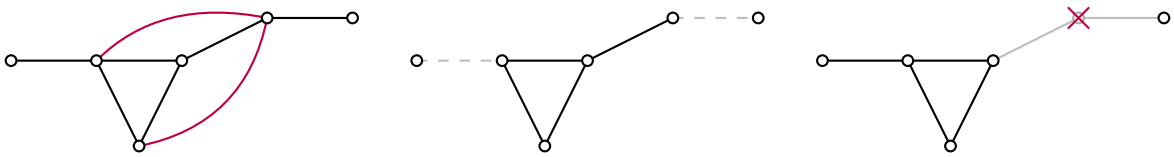

Figure 1.7: A graph with a split completion, a split deletion, and an induced split subgraph.

Many classical problems on graphs are hard in general, but can be solved by polynomial algorithms as long as the input graph belongs to an appropriate graph class. This is for example the case for the *maximum clique* problem, asking for the size of the biggest clique in the graph, or for the *colouring* problem: both are proved to be NP-hard (see the excellent book by Garey & Johnson [47] for more NP-hardness and NP-complete information), but become polynomial on chordal graphs and all their subclasses [49]. In this kind of situation, completions and deletions turn out to be a useful tool.

Consider a problem that is hard in general but can be solved in polynomial time on a certain graph class $\mathcal{P}$. For a graph G, if it does not already fulfil the property $\mathcal{P}$, the idea is to compute a minimal $\mathcal{P}$-completion (or a minimal $\mathcal{P}$-deletion, depending on what is considered) of G and solve the problem in polynomial time on this new graph. Of course, the solution obtained this way is maybe not the real solution of the problem on input G, but it can be seen as an "approximation" – without any guarantees, though – of the real solution, that can be found in polynomial time.

When it comes to more structural aspects of graph theory, some questions about graph decompositions can be addressed. In fact, some well-known graph decompositions are defined from the graph classes themselves. The most well-known example of this is certainly the tree-decomposition. Since this decomposition is closely related to a very important graph parameter called *treewidth*, it permits to solve a large number of problems in polynomial time on chordal graphs, provided that the "width" is bounded. Some other graph parameters are strongly linked to graph properties: let us cite the *treedepth* and the class of *Trivially Perfect* graphs – which correspond to chordal cographs – or the *twin-width*, closely related to cographs.

Because it is usually hard to know exactly how many fill edges are required in order to give the appropriate property to an input graph, being able to compute minimal completions into the appropriate class would be very useful. Indeed, it is easier to compute a decomposition for a completed graph, and consider it as a decomposition, maybe not optimal, of the input graph.

Another motivation of the study of completions and deletions in a given graph class is the structure present in many real-world datasets. Some of them have a structure that is very close to an existing and well-studied graph class such as cographs [33]. It is of interest to know how far these graphs acquired through real data are from the considered graph class: once again, this is a difficult problem that can be "approximated" by completions and deletions. Graph completions into a property could also be used to repair a graph that was known to have this specific property, but has been transferred via a communication network in which some information has been lost.

Graph completions and deletions are in fact particular cases of the more general framework of *graph modification* problems. Another problem that is often studied in this framework is the *graph edition* problem. Computing an edition of a graph G into a class $\mathcal{P}$ consists in finding a set of two-element subsets of its vertex set V, that can be edges or not, that have to be modified in order to make G obtain the property $\mathcal{P}$. The considered modification is a very simple one: for $u, v \in V$, if uv is an edge in G, then it is removed, else it is added to E. Like completions and deletions, the graph editions that are studied in practice are often the ones that are minimal for inclusion, that is to say, for which the set of modifications is minimal with respect to inclusion [67].

Although convinced of the interest of graph edition problems [32, 40], we will not investigate them further in this thesis. Their nature is slightly different from that of the two other graph modification problems, and very little is known about graph editions in the world of enumeration. Anyway, we will focus here on the graph completions and deletions problems.

Chapter 2

Enumeration problems on graphs

This chapter is intended as an introduction to enumeration problems and the measure of their complexities. Other sources of interest on this topic include (but are not limited to) the works of Defrain [35], Mary [64], and Strozecki [79, 80]. We give here some elements to state our problems in the context of enumeration.

The graph problems in which we are interested are part of the more general family of *enumeration problems*. Therefore, let us first have a closer look on this special kind of problems, their definition, and the ways to measure the complexity of the corresponding algorithms. Then, we will have an overview of the existing efficient enumeration techniques. In the last section of the present chapter, we state more precisely which type of problems will be studied.

2.1 Definitions and models

In this section, we present more formally the general concepts and the specificities of enumeration.

2.1.1 Enumeration problems: definition

Let X be a set and $\mathcal{R}$ be a binary relation on X. For one $x \in X$, called the *input* or *instance*, let us denote by $\mathcal{R}(x) := \{y \in X \mid (x, y) \in \mathcal{R}\}$ the set of all y such that $(x, y) \in \mathcal{R}$. The elements of $\mathcal{R}(x)$ are called *solutions*, or *outputs*.

Definition 2.1 (Enumeration problem)**.** *The* enumeration problem *associated with the binary relation $\mathcal{R}$ on X takes in input an $x \in X$, and outputs the set $\mathcal{R}(x)$.*

An algorithm which solves an enumeration problem is called an *enumeration algorithm*. More precisely, an enumeration algorithm called on an input x outputs an ordered sequence $y_1, \dots, y_N$ such that $\{y_1, \dots, y_N\} = \mathcal{R}(x)$. In particular, enumeration algorithms generate each solution of the problem exactly once.

It is worth noticing that enumeration generalises recognition: we must be able to certify that all outputs are indeed solutions. This is why we will focus on problems for which recognition is possible in polynomial time.

Computational model. The computational model we use is the Random Access Machine (RAM) model, equipped with the four basic operations (comparison, addition, subtraction, multiplication), and a special instruction `output(i,j)` – different from the classical `return` instruction – which displays the content of registers $R_i, R_{i+1}, \dots, R_j$ in constant time without stopping the computation. This is the model commonly used for enumeration problems [3, 35, 79, 80]. In particular, it permits to access an exponential amount of memory in polynomial time, using the appropriate dictionary data structures.

2.1.2 Complexity of enumeration algorithms

Given an enumeration problem, the number of solutions that have to be outputted can be very large. Indeed, it is not rare for such a problem to admit an exponential number of solutions. To cite a simple example, it has been shown that a graph on n vertices can have as many as $3^{n/3}$ inclusion-wise maximal stable sets [66]. In such a case, we still want to generate all solutions in a reasonable time, but there is obviously no hope for outputting an exponential number of objects in polynomial time, since we need at least the time of writing them. It is therefore no longer relevant to measure the running time of an enumeration algorithm with the classical complexity classes such as P or NP. Knowing this, how can we adapt the notion of efficiency to an algorithm which outputs a potentially exponential number of solutions?

To answer this question, two main approaches have been proposed in the literature. One of them, called *input-sensitive*, consists in finding the fastest possible exact exponential algorithms, since the considered problems are exponential in the input size. The other one, *output-sensitive*, assumes that, since all solutions have to be generated anyway, a good measure of efficiency for enumeration algorithms should take this into account. The two approaches give results of different types, and in that sense they are complementary.

In this thesis, we will always measure the complexity in an output-sensitive way. However, for the sake of completeness, both approaches are mentioned here.

Input-sensitive approach. This approach consists in measuring the running time of enumeration algorithms in terms of the input size only. In this case, algorithms usually

run in exponential time, and research typically focuses on lowering the exponent.

Since the running time of an enumeration algorithm is necessarily lower-bounded by the number of solutions, expressing its complexity in terms of the input size can give useful insights for the number of solutions that have to be outputted. In fact, many known input-sensitive enumeration algorithms were obtained by bounding tightly the number of objects to enumerate [66, 30]. Another advantage of measuring the complexity of enumeration algorithms in terms of the input size only is to give guarantees on the total running time.

However, if the number of solutions to output is exponentially larger than the input size, an enumeration algorithm will not be considered "efficient" in the usual sense, even if it needs a constant time per solution. Indeed, knowing that there exists an exponential enumeration algorithm gives in fact no insight on the real efficiency of the algorithm: is it exponential in the input size because there is an exponential number of solutions, or is it exponential in the number of solutions?

In this sense, the input sensitive-approach seems therefore not the most satisfactory to capture the notion of efficiency for enumeration algorithms.

Output-sensitive approach. The output-sensitive approach is based on the idea that complexity measures for enumeration problems need to take into account the potentially large number of solutions they can have. To measure its efficiency, the running time of an algorithm is then bounded by a function of the input size n *and* the number N of solutions. Since the output-sensitive complexity measures take into account the large number of solutions, they allow us to capture a notion of running time *per solution* and to compare the complexities of two problems that do not possess the same number of solutions.

This is the point of view adopted in all the remaining of this thesis: all algorithms presented here have their complexities measured in terms of the input size and the output size.

When dealing with enumeration, as for many other domains, our aim is to design efficient algorithms. In classical complexity theory, the algorithms that are regarded as "efficient" are polynomial algorithms. This raises a question: since polynomial algorithms have no sense in general for enumeration, how can we capture the essence of efficiency in this case? To answer this question, three complexity measures have been proposed by Johnson, Yannakakis & Papadimitriou in the 1980s [57].

Output-polynomial. An algorithm is said to be *output-polynomial* if its total running time is bounded by a polynomial both in the input size n *and* the number of solutions N that have to be outputted. This ensures every solution will eventually be generated in a reasonable amount of time. This complexity measure is also called "Polynomial total time". It is the natural translation of polynomial time

complexity into the field of enumeration problems. An illustration of the running time of an output-polynomial algorithm is presented in Figure 2.1.

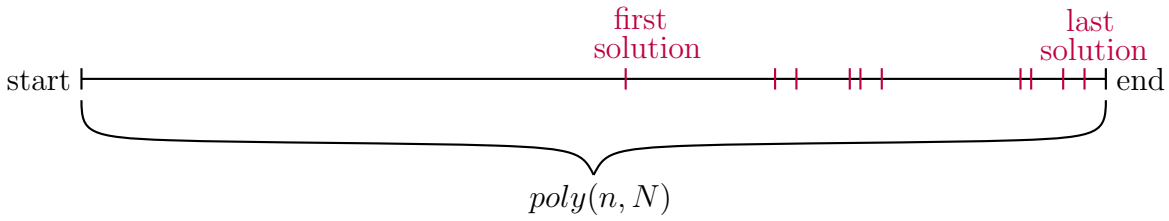

Figure 2.1: Execution time of an output-polynomial algorithm. A vertical line indicates the output of a solution.

The issue with output-polynomial algorithms is mostly that there is no guarantee on the time needed to generate the first solution. A large amount of time (polynomial in N but still exponential in the input size n) could be necessary before outputting just one solution, which may not be satisfactory for many practical cases. We try to highlight this phenomenon on Figure 2.1.

Incremental-polynomial. To guarantee a fast access to the first generated solutions, the time needed by an *incremental-polynomial* algorithm to output a solution is polynomial in the input size n and the number of solutions already found. In other words, for each $i \leqslant N$, the time needed between the ith solution and the next one is bounded by a polynomial in n and i. To help the reader visualise the behaviour of an incremental-polynomial algorithm in terms of running time, it is schematised in Figure 2.2.

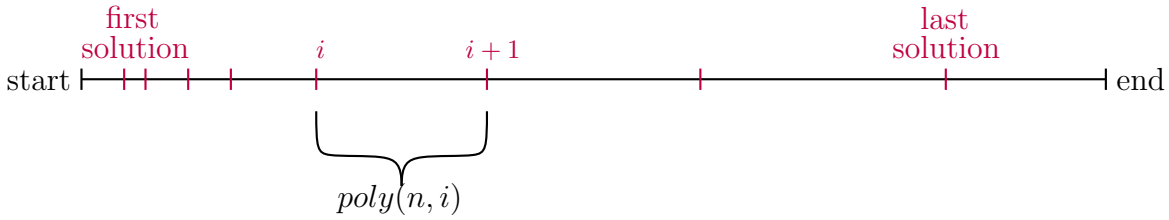

Figure 2.2: Execution time of an incremental-polynomial algorithm. A vertical line indicates the output of a solution.

Incremental-polynomial algorithms are obviously output-polynomial. However, an incremental-polynomial algorithm ensures in addition that solutions will be outputted all along the execution, and not only at the end, as it could be the case for output-polynomial algorithms. Moreover, with such an algorithm the first solutions are outputted quickly, even if the computation of the following ones

can take more and more time. Such guarantees are therefore of better use for applications.

Polynomial delay. An algorithm has *polynomial delay* if the time needed before the first solution, between two consecutive solutions (regardless of the number of already generated solutions), and to stop after the last solution, is polynomial *only* in the input size n. To visualise the difference between polynomial delay and the other output-sensitive complexity measures presented here, the running time of a polynomial delay algorithm is illustrated in Figure 2.3. Algorithms running with polynomial delay are the most efficient among those considered in this thesis.

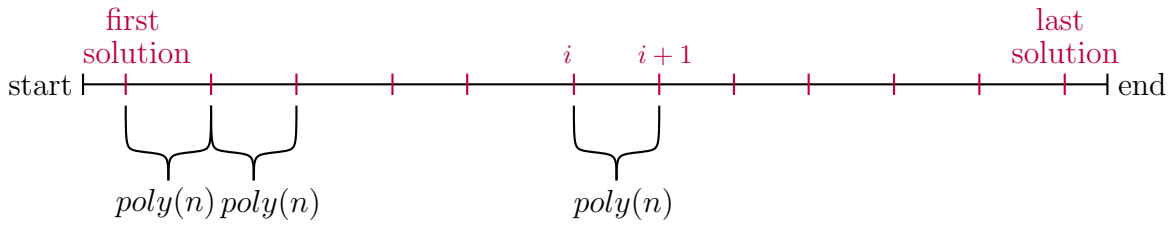

Figure 2.3: Execution time of a polynomial delay algorithm. A vertical line indicates the output of a solution.

Necessarily, polynomial delay algorithms are incremental-polynomial, and therefore output-polynomial, linear in the number of solutions. The main advantage of polynomial delay algorithms is to completely remove the dependence on the number of solutions already found. This way, solutions are outputted quickly all along the algorithm's execution.

The inclusions between the three historical complexity classes that have just been presented are schematised in Figure 2.4. But these complexity measures are not the only ones that are considered in practice.

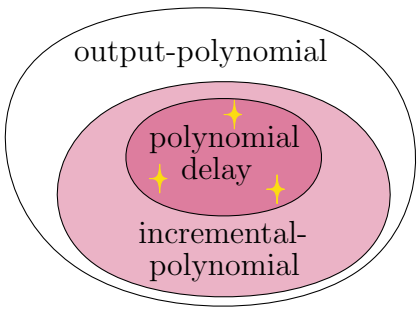

Figure 2.4: Inclusions between the three complexity classes *output-polynomial*, *incremental-polynomial*, and *polynomial delay*.

In fact, polynomial delay is not the best complexity we could hope for in all cases. For some problems, polynomial delay could even be lowered to *constant delay*, meaning that the time needed between two consecutive outputs is bounded by a constant. This complexity cannot be achieved for all problems [80], but sometimes the problem's structure allows to use a particularly well-suited enumeration technique, such as *Gray codes*.

When the solutions can be represented as words on a finite alphabet, it is possible to enumerate them *via Gray codes*. The idea of enumeration with Gray codes is to generate all the solutions in such a way that the number of different characters between two consecutive solutions is exactly one. Therefore, in such a case it is not even necessary to output the whole new solution when one is found, because outputting the difference with the previous one suffices, thus achieving constant delay for problems that have solutions of polynomial size. Moreover, it is only necessary to keep in memory the last generated solution, hence using only polynomial space.

Gray codes have been introduced for the first time in 1941 as *reflected binary codes* in order to enumerate binary integers on n bits [78], but they can also be used to enumerate such various objects as subsets of a given set, permutations of a set [58], or configurations of Hanoi Towers. Many more applications of Gray codes can be found in Ruskey's (unpublished) book [76].

Gray codes are not the only mean to achieve constant delay for a given problem. Let us mention briefly the *Fisher-Yates shuffle* [39], with its constant-delay variant, for the generation element by element of a random permutation. This technique is especially used for generating a permutation α of a large set [19]. With this method, for any i, the time needed between the output of $\alpha(i)$ and $\alpha(i+1)$ is constant.

We have surveyed some of the most classical output-sensitive complexity measures. In fact, many more notions exist for output-sensitive algorithms [31, 80], but they will not be encountered here.

Yet, we do not know output-polynomial algorithms for all enumeration problems. It is even proven that, unless $\mathsf{P} = \mathsf{NP}$, there are enumeration problems for which no such efficient algorithm can be found [14, 62]. In some cases, what happens is that there exists a polynomial number of "trivial" solutions, which can be found in polynomial time, but the time necessary to find another solution is exponential in the input size.

This is for example the case with the so-called *minimal rainbow induced subgraphs* problem, taking in input a graph whose edges are assigned one or more colours and asking for the list of all minimal induced subgraphs containing at least an edge of each colour [38]. In some instances, finding a non-trivial solution is equivalent to finding a truth assignment of a SAT formula, therefore impossible in polynomial time if $\mathsf{P} \neq \mathsf{NP}$.

An enumeration algorithm is said to be *output-quasi-polynomial* if it runs in total time $\mathcal{O}(2^{\mathrm{polylog}(n+N)})$, where n denotes the input size, and N the number of solutions.

When an output-polynomial algorithm cannot be found, it is often desirable to provide an output-quasi-polynomial algorithm that solves the problem [64, 35].

Space complexity. The space complexity of enumeration algorithms is also a challenge: many enumeration techniques require to store the generated solutions in memory, in order to check if the newly discovered solution is really new, or if it has already been found in the past. In general, when the number of solutions is exponential, this naturally leads to exponential-space algorithms. However, in some cases it is possible to avoid storing all solutions in memory, therefore reducing the space complexity of the algorithm to make it polynomial in the input size.

2.1.3 Classical enumeration problems

Among all enumeration problems, some are widely known and studied, mostly because they are very general. These have been surveyed in previous work [64]. We give here a brief overview of some of the most classical enumeration problems, without aiming at being exhaustive.

Independent set systems and dependent set systems. A set family $\mathcal{I}$ on a ground set E is called an *independent set system* if $\varnothing \in \mathcal{I}$ and for any two $I_1, I_2 \subseteq E$, if $I_1 \in \mathcal{I}$ and $I_2 \subseteq I_1$, then $I_2 \in \mathcal{I}$. In other words, an independent set system is a family of sets closed under inclusion.

Independent set systems are far from being rare in enumeration problems. In particular, many set families naturally have this property: in graphs, some very classical objects such as stable sets, cliques, or matchings define independent set systems. Due to their omnipresence, these set families are interesting objects to enumerate.

In fact, when dealing with independent set systems, it often suffices to generate all the independent sets that are inclusion-wise maximal, since all sets included in any of them are by definition elements of the family. It was shown by Lawler, Lenstra & Rinnooy Kan [62] that in general, if the set system is given by a constant-time membership oracle, there does unfortunately not exist an output-polynomial algorithm enumerating all the maximal elements of an independent set system, unless $\mathsf{P} = \mathsf{NP}$. However, they also proved that such an algorithm exists for many practical particular cases, as it will be shown in the next section.

Independent set systems are closely related to *dependent set systems*, that are set families closed under taking supersets. In the case of dependent set systems, the interesting elements to enumerate are the inclusion-wise *minimal* sets belonging to the family. Some dependent set systems are very famous: dominating sets, Vertex Covers, transversals of an hypergraph...

Enumeration of minimal transversals of an hypergraph. Hypergraphs are a generalisation of graphs with edges of arbitrary size. An hypergraph can be seen as a family of sets $\mathcal{E}$, the hyperedges, on a set of vertices V. A *transversal* of an hypergraph $\mathcal{H} = (V, \mathcal{E})$ is a set of vertices T intersecting every hyperedge of $\mathcal{H}$, that is to say, for any $E \in \mathcal{E}$, $T \cap E \neq \varnothing$. The transversals of an hypergraph are also referred to as *hitting sets*.

The associated enumeration problem is the following: given a hypergraph $\mathcal{H}$, generate all minimal transversals of it. Equivalently, it is sometimes asked to return a hypergraph whose hyperedges are the minimal transversals of $\mathcal{H}$. The minimal transversals enumeration problem is so general that numerous enumeration problems (corresponding to dependent set systems) can be seen as particular cases of it.

For example, the *dominating sets* of a graph G – the sets of vertices D such that every vertex belongs to D or has a neighbour in D – can be seen as the transversals of an hypergraph whose hyperedges are the closed neighbourhoods of all the vertices of G. Then, enumerating the minimal dominating sets of a graph [60] is a particular case of the minimal transversals enumeration. In another direction, we can look at the *Feedback Vertex Set* problem, which asks for the minimal sets of vertices whose removal makes the input graph acyclic. Then, enumerating all feedback vertex sets of G [77] is equivalent to enumerating all minimal transversal of the hypergraph on the vertex set V whose hyperedges are the sets of vertices inducing cycles in G.

The current best algorithm for the enumeration of minimal transversals of an hypergraph dates back to the 1990s. It is due to Fredman & Khachiyan [45] and runs in output-quasi-polynomial time. Knowing if all minimal transversals of an hypergraph can be enumerated in output-polynomial time is a famous, long-lasting open question [41]. This question is in fact not restricted to hypergraphs: several formalisms capture the same problem [6].

2.2 Algorithmic methods for enumeration

To generate all solutions of a problem, the principle is often to give a structure to the space of solutions, and then explore it. Each time a new solution is found, it is outputted. Sometimes, the structure is given to the space of solutions *via ad-hoc* methods. This is for example the case with the ingenious algorithm proposed by Carmeli *et al.* [18] to generate all minimal chordal completions of a graph. In Chapter 3, we will give another, simpler example of such an *ad-hoc* algorithm when enumerating minimal split completions. However, there also exist generic methods that can apply to a wide range of problems. Some of these general frameworks are known to give good complexity results, but they are not always applicable. Among the most popular techniques used to devise enumeration algorithms, let us cite *Flashlight Search*, maximal independent set generation, the *Input Restricted Problem* approach, *Reverse Search*, and *Proximity*

Search.

In the whole section, we state a general enumeration problem as follows: given a ground set E, with $n = |E|$, and a property $\mathcal{P}$, generate all (inclusion-wise maximal) subsets of E which satisfy property $\mathcal{P}$.

2.2.1 Flashlight Search

When dealing with (maximal) subset enumeration, a classical technique sometimes called Backtrack [72], Binary Partition [25, 60], or *Flashlight Search* [15, 16] naturally arises. Let us insist here on the fact that the property $\mathcal{P}$ under consideration – as it will always be the case in the present work – is hereditary.

The Flashlight algorithm consists in choosing an arbitrary ordering $e_1, \ldots, e_n$ on the elements of E, and considering them in order. To begin, we have to decide whether there exists a solution containing e_1, and if there exists a solution excluding e_1. Then, each time a solution S exists up to elements $e_1, \ldots, e_i$, we have to decide if there exists a solution compatible with S and containing e_{i+1}, and a solution compatible with S and excluding e_{i+1}. If not, the branch is no longer explored. This way, we build a binary tree rooted at $\varnothing$ such that each internal node at level i represents a subset of $\{e_1, \ldots, e_i\}$ satisfying $\mathcal{P}$, and the leaves correspond exactly to the maximal solutions. An example of the tree explored by Flashlight Search is presented in Figure 2.5. The tree is searched while being constructed, outputting a solution each time a new one is found. The decision problem solved at each step of the computation is called an *extension problem*.

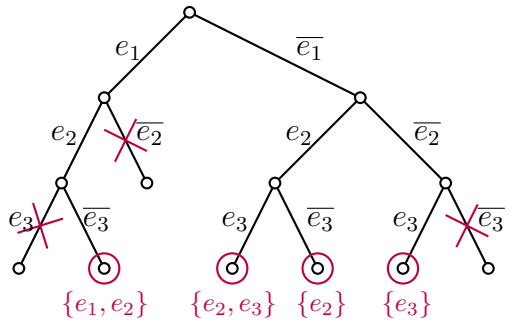

Figure 2.5: Illustration of the tree explored by Flashlight Search.

Flashlight Search is a powerful technique: if the extension problem is solvable in polynomial time, it yields a polynomial delay algorithm for the enumeration of all (maximal) subsets of E satisfying $\mathcal{P}$ [72]. Unfortunately, the extension problems associated with enumeration problems are not polynomial in general: they will be addressed in detail for maximal subgraphs in Chapter 4.

When this technique was introduced by Read & Tarjan [72], it was used for example to enumerate all spanning trees of a connected graph. Indeed, knowing if a set of edges can be extended into a spanning tree is rather easy. In fact, it suffices to check if this set of edges induces an acyclic graph: if not, it cannot be extended into a spanning tree. Forbidding some edges to belong to a solution does not make the problem much harder, as we will see in Chapter 4. Consequently, Flashlight Search yields a polynomial delay polynomial space algorithm for the enumeration of spanning trees of a connected graph.

Maximal subgraphs are not the only application of Flashlight Search: due to its generality, this technique can be applied to many different problems. For example, it is used to enumerate efficiently the truth assignments of a logic formula under certain conditions [15].

2.2.2 Maximal independent set generation

The goal here is to generate all inclusion-wise maximal elements of an independent set system. If $\mathcal{P}$ is a *hereditary* property, that is, closed under inclusion, then the family of all subsets of E satisfying $\mathcal{P}$ is an independent set system $\mathcal{I}$. Generating all maximal subsets of E satisfying $\mathcal{P}$ is thus the same as generating the maximal elements of $\mathcal{I}$.

Let us take a look on a technique designed by Lawler, Lenstra & Rinnooy Kan [62] to enumerate all maximal elements of an independent set system, generalising the work of Paull & Unger [71]. They assume that the set family under consideration is given by a membership oracle – a function answering TRUE on a set if it belongs to the independent set system, FALSE otherwise – whose running time is denoted by c. Then, if the ground set is arbitrarily ordered $E = \{e_1, e_2, \ldots, e_n\}$, the algorithm proceeds incrementally, following the ordering. For any $i \leqslant n$, denote by $\mathcal{I}_i$ the family of all independent sets included in $\{e_1, \ldots, e_i\}$. They prove the following theorem.

Theorem 2.2 ([62], Theorem 4)**.** *It is possible to enumerate the maximal elements of an independent set system $\mathcal{I}$ in total time polynomial in n, N and c, provided that for any $j \leqslant n$, and for any $I \in \mathcal{I}_{j-1}$, the maximal independent sets of $I \cup \{e_j\}$ can be found in polynomial time.*

The general algorithm they give for maximal independent set generation is presented here as Algorithm 2.1. If it is possible to generate all maximal independent sets of $I \cup \{e_i\}$ in polynomial time (line 12), then this Algorithm runs in polynomial total time $\mathcal{O}(ncN + n^2cN^2 + n^2N^3)$ [62]. However, this classical algorithm needs a time polynomial in N to output a first solution. Therefore, its time complexity is output-polynomial but not incremental-polynomial.

Due to the large presence of independent set systems among optimisation problems, the method introduced by Lawler, Lenstra & Rinnooy Kan can be used in various contexts. For example, it permits to generate all maximal feasible solutions to a

Algorithm 2.1: Maximal independent set generation

input : An independent set system $\mathcal{I}$ on E (given by a membership oracle in time c)
output : All maximal elements of $\mathcal{I}$

```
1  S := ∅;        /* the set of all partial solutions */;
2  i = 1;
3  while i ⩽ n do
4  |  S :=EXTEND (S, e_i, I);
5  |  increment i;
6  return S;

   Function EXTEND (S, e_i, I):
7  |  T := ∅;
8  |  foreach I ∈ S do
9  |  |  if I ∪ {e_i} ∈ I then
10 |  |  |  T := T ∪ {I ∪ {e_i}};        /* I ∪ {e_i} is maximal in I_i */;
11 |  |  else
12 |  |  |  foreach J ∈ I maximal within I ∪ {e_i} do
13 |  |  |  |  if J is maximal in I_i then
14 |  |  |  |  |  T := T ∪ {J};        /* do not add if J ∈ T already */;
15 |  return T;
```

knapsack problem by extending partial solutions into maximal ones, or to enumerate all maximal feasible solutions of an inequality system [62].

It can also be used to solve enumeration problems on graphs. Lawler *et al.* applied their technique to the enumeration of all maximal complete k-partite subgraphs, that is, subgraphs formed of k disjoint sets, and where two vertices are adjacent if and only if they do not belong to the same set. Therefore, this technique gives an output-polynomial algorithm for the enumeration of maximal complete k-partite subgraphs.

The setting being very general, Algorithm 2.1 can be applied to many different problems. It might not have the optimal complexity in all cases, but it still provides an efficient way to solve a large set of enumeration problems.

2.2.3 Reverse Search

In 1992, Avis & Fukuda [2] designed a general framework for enumeration problems, called *Reverse Search*. At the time, several works on enumeration were using similar techniques. For example, the well-known algorithm of Tsukiyama *et al.* for the enumeration of maximal cliques of a graph [84] and the algorithm of Johnson *et al.* for the enumeration of maximal elements of an independent set system with polynomial

delay [57] used similar methods. Those techniques were designed to improve the existing ones such as the maximal independent set generation of Lawler *et al.* [62]. When Reverse Search was made explicit, it unified these frameworks.

Reverse Search is designed explicitly for polynomial delay and polynomial space. It is the "reverse" approach of *local search*, a technique often used in optimisation which consists in navigating the space of solutions, transforming a solution into a better one with local changes, until a local optimum is found. The principle of Reverse Search is the following: build a *rooted* spanning arborescence on the solution space by defining for each solution S its *parent* $\textsc{Parent}(S)$, and explore the arborescence in a DFS manner, keeping in memory the parent of the current solution. The function $\textsc{Parent}$ is therefore defined on the solution space, apart from the root(s). To illustrate the idea, let us take a look at Figure 2.6.

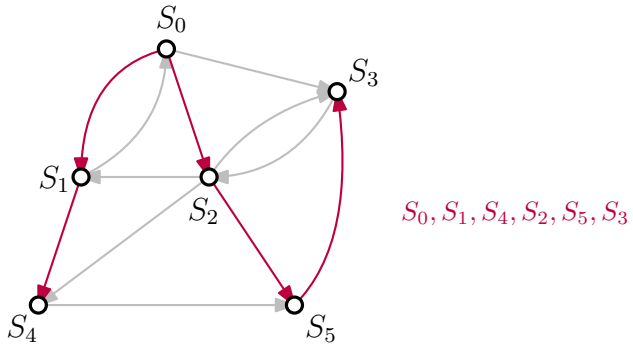

Figure 2.6: An example of the arborescence induced by the function $\textsc{Parent}$ on the solution space. Reverse Search consists in enumerating the solutions by following this arborescence.

More precisely, the arborescence is defined by the following information:

- a root solution S_0 (or a set of root solutions if the arborescence is a forest rather than a tree);
- a function $\textsc{neighbours}$ for generating in polynomial time all neighbouring solutions of the current one in the solution space: in Avis & Fukuda's original paper [2], the neighbours of a solution are computed by the $\textsc{LocalSearch}$ procedure;
- a function $\textsc{Parent}$ to decide for each solution which one is its parent, with the property that following the parent relation, one will eventually reach S_0 (or any root solution). This $\textsc{Parent}$ function allows us to define for each solution S the set of its children, that is, $\textsc{Children}(S) = \{S' \in \textsc{neighbours}(S) \mid \textsc{Parent}(S') = S\}$.

Note that the children of a solution form a subset of its neighbours. Since each solution (except the root) has one parent, the previous requirements ensure that each solution is generated at most once.

The general Reverse Search algorithm is quite simple: it is presented here as Algorithm 2.2.

Algorithm 2.2: The general Reverse Search algorithm

input : An enumeration problem, together with a polynomially computable Parent function
output : All solutions of the problem

```
1 S_0 := a particular "root" solution;
2 Call enum (S_0);
  Function enum (S):
3     Output S if recursion depth is even;
4     foreach S' ∈ Children (S) do
5         Call enum (S');
6     Output S if recursion depth is odd;
```

Delay. If the set of children of a solution can be computed in polynomial time (in particular, if any solution has a polynomial number of neighbours), then Algorithm 2.2 has polynomial delay. But to guarantee this delay, we need to be careful about outputting the solutions at the right time.

When exploring the solution space with the Parent relation, one may have to explore a path whose length is exponential in the input size. Therefore, if each new solution is outputted at the time it is found, then it might be necessary to backtrack during an exponential time when the whole branch has been explored. This leads to an exponential delay between two solutions, which is not very efficient. However, a classical trick, sometimes called the *Alternative Output Method* [85] is to output the new solutions at the beginning of the recursion step if the recursion depth is even (line 3), and at the end if the depth is odd. This way, the time spent between two consecutive solutions can be lengthened, but solutions are also outputted when backtracking the DFS, preventing the algorithm to run during an exponential time without producing any new solution.

Because the structure given to the solution space is an arborescence, it is no longer necessary to keep in memory all solutions that have already been found. Being able to compute the parent of each solution in polynomial time guarantees the knowledge of the current state. Since each solution (apart from the root solutions) has one parent, there is exactly one way to reach this solution by following the arborescence. Therefore, there is no need to check: whenever a new solution is generated, we know for sure it has never been outputted before. Hence, no memory is required to store an exponential number of solutions: Algorithm 2.2 runs in polynomial space.

The Reverse Search technique has proved efficient on a wide variety of problems. In their original article [2], Avis & Fukuda already applied it in many situations, to enumerate such various objects as cells in an arrangement, topological orderings, or connected induced subgraphs. In fact, it is the reference technique when designing polynomial delay polynomial space enumeration algorithms [11, 24].

For example, this method allows us to generate in polynomial delay and polynomial space all spanning trees of a connected graph [2]. In this case, the root solution is the lexicographically smallest spanning tree, according to an arbitrary order on the edges. To generate a neighbour of a solution, it suffices to add an edge and break the cycle that has been created by removing another edge. Finally, the parent of a solution is defined as its lexicographically smallest neighbour.

In the same article, Reverse Search has also been used to enumerate efficiently all triangulations of a set of points in the plane: the neighbours of a solution are obtained by "flipping" an edge, that is, exchanging diagonals in a quadrilateral.

2.2.4 Input Restricted Problem

The problem considered here is the enumeration of maximal induced subgraphs of a graph $G = (V, E)$, with $n = |V|$, that verify a *hereditary* property $\mathcal{P}$. For this problem, Cohen, Kimelfeld & Sagiv [22] devised an interesting framework. They manage to link the complexity of enumerating maximal induced subgraphs that verifies a hereditary property $\mathcal{P}$ to the complexity of a so-called *Input Restricted Problem*.

Given an induced subgraph H of G that satisfies $\mathcal{P}$ and an additional vertex $v \in V$ not in H, the *Input Restricted Problem* consists in asking for the list of all maximal induced subgraphs of $H + v$ fulfilling property $\mathcal{P}$.

The idea behind this technique is a very classical one: starting from any solution of the problem, new solutions are found from the already generated ones. This way, an oriented *supergraph of solutions* is built, whose vertices are all the maximal induced subgraphs we want to enumerate, and an arc is put from a solution S to a solution S' if there exists a vertex v such that S' is obtained *via* a maximal induced $\mathcal{P}$-subgraph of $S + v$. Since the property is hereditary, it is easy to find a maximal induced $\mathcal{P}$-subgraph: it suffices to build it greedily, starting from the empty set and adding vertices in any order. One can then explore the supergraph of solutions while constructing it and output each new generated solution.

Suppose the Input Restricted Problem can be solved by a property-specific procedure genRestrHered(H, v), that outputs all the maximal induced $\mathcal{P}$-subgraphs of $H + v$ for any maximal induced $\mathcal{P}$-subgraph H of G and any vertex $v \in V$ not in H. Then, Algorithm 2.3 can be used to enumerate all the maximal induced $\mathcal{P}$-subgraphs. Note that a maximal induced $\mathcal{P}$-subgraph of $H + v$ is not necessarily maximal in G: that is why it needs to be maximised (line 7) in order to find a solution. Since the property

under consideration is supposed to be hereditary, a greedy maximisation is well-suited for this purpose.

Algorithm 2.3: Enumeration of the maximal induced $\mathcal{P}$-subgraphs *via* the Input Restricted Problem

input : A graph G
output : All maximal induced $\mathcal{P}$-subgraphs of G

```
1  H := any maximal P-subgraph of G;
2  S := {H};        /* the set of solutions already found */;
3  Output H;
4  Call recursiveGen (G, H, S);

   Function recursiveGen (G, H, S):
5      foreach v ∈ V(G) \ V(H) do
6          foreach H' ∈ genRestrHered (H, v) do
7              Maximise H' greedily within G;
8              if H' ∉ S then
9                  Output H' if recursion depth is even;
10                 S := S ∪ H';
11                 Call recursiveGen (G, H', S);
12                 Output H' if recursion depth is odd;
```

The Input Restricted Problem technique is different from the maximal independent set generation framework. For maximal independent set generation, the technique discussed above generates partial solutions all along the execution, and finds all the maximal independent sets at the same time at step n. On the other hand, in the algorithm proposed by Cohen *et al.* (here presented as Algorithm 2.3), when a new partial solution is found, it is immediately maximised with a greedy maximisation function, and outputted. This way, maximal induced subgraphs are produced all along the execution of the algorithm.

When guarantees are known on the complexity of genRestrHered, other complexity guarantees can be deduced for the whole enumeration algorithm. Based on this technique, the authors proved the following result.

Theorem 2.3 ([22])**.** *For all hereditary or connected-hereditary properties $\mathcal{P}$, the maximal $\mathcal{P}$-subgraphs can be enumerated:*

- *in output-polynomial time if and only if the input restricted problem can be solved in output-polynomial time;*
- *in incremental-polynomial time if and only if the input restricted problem can be solved in incremental-polynomial time;*

- *with polynomial delay if the input restricted problem can be solved in polynomial time.*

The authors also prove results on space usage. In particular, for a hereditary property $\mathcal{P}$, if the input restricted problem can be solved in polynomial space, then Algorithm 2.3 can be improved to run in polynomial space.

There are several "natural" problems for which the input restricted problem is polynomial. Let us cite for example the maximal clique enumeration problem. Given a maximal clique K of a graph G and a vertex v that does not belong to K, there are only two maximal cliques in the graph induced by $K \cup \{v\}$. These are K and $N_K(v) \cup \{v\}$ (which is a clique since all vertices are connected in K). Therefore, the input restricted problem has only two solutions and can be solved in time polynomial in the size of G: Theorem 2.3 ensures that all maximal cliques of G can be enumerated with polynomial delay. In this case, polynomial space is achieved as well.

The approach proposed by Cohen *et al.* is very efficient when it can be applied. However, it is not always the case that the input restricted problem is solvable in polynomial time (an example will be given in Chapter 5).

2.2.5 Proximity Search

Proximity Search has been recently introduced by Conte & Uno [26]. The main advantage of this technique is its generality. Proximity Search was designed explicitly for maximal subset enumeration, producing polynomial delay algorithms for a wide range of problems [23]. It overcomes some limitations of the Input Restricted Problem approach without giving such strong guarantees, in particular for space usage.

As several other enumeration paradigms [2, 22], Proximity Search relies on an organised walk on the space of solutions. The general idea is very similar to the one used with the Input Restricted Problem, or with Reverse Search: build an oriented *supergraph of solutions*, whose vertices are the desired solutions of the problem, and arcs are put between them according to a problem-specific operation that permits us to transform a solution into some others in polynomial time. This *transition function* cannot be reversed in general.

The novelty of Proximity Search is the introduction of a *proximity* measure between two solutions, to provide the solution space with a metric, and be able to measure "how close" two solutions are. In order to define the proximity measure, solutions are considered here as ordered subsets of E. This allows us to consider the *prefixes* of a solution: if $S = e_1, \ldots, e_k$ is an ordered maximal subset of E satisfying property $\mathcal{P}$, the prefixes of S are all the $e_1, \ldots, e_i$ for all $i \leqslant k$. Note that each solution has its own ordering, and two different solutions can have completely unrelated orderings *a priori*.

Definition 2.4 (Proximity)**.** *The* proximity $S \widetilde{\cap} T$ *between two solutions* S *and* T *is the longest prefix of* T *that is contained in* S*.*

This notion of proximity does not need to induce a distance in the mathematical sense[1] because it is not symmetric in general, but it must satisfy the crucial property

$$|S \widetilde{\cap} S'| = |S'| \Rightarrow S = S'.$$

There are three major requirements to apply Proximity Search.

- An *ordering scheme*, that is, a function π which associates to each $\mathcal{P}$-subset of E a total ordering of its elements. For a subset $S \subseteq E$, $\pi(S)$ is called the *canonical ordering* of S.
- A function NEIGHBOURS, computing all (out-)neighbours of a given solution in time polynomial in n. In particular, every solution must have a polynomial number of neighbours. This neighbouring function relies on a transition operation that, given a solution S, can produce other, different maximal $\mathcal{P}$-subsets of E. In any case, the transition operation is highly specific to the considered problem.
- A maximisation function COMPLETE, taking in input a partial solution – a $\mathcal{P}$-subset of E – and completing it into a maximal one. The use of that function is mostly hidden in the computation of NEIGHBOURS. Therefore, the function COMPLETE needs to be computable in polynomial time.

The goal is then to show that the super-graph of solutions is strongly connected. To see this, it suffices to prove that for any given solution, the proximity to a target solution can be increased at each step. More precisely, given a solution S and a target solution S^*, we want to prove that there exists another solution among the neighbours of S that has a higher proximity with S^*. This way, the search is ensured to visit every solution at least once.

Definition 2.5 (Proximity searchable)**.** *Let us consider an enumeration problem, together with an ordering scheme π and a polynomial-time computable neighbouring function* NEIGHBOURS*. The problem is called* proximity searchable *if one solution can be found in polynomial time and, given any two solutions S and S^* such that $S \neq S^*$, there exists $T \in \text{NEIGHBOURS}(S)$ such that $|T \widetilde{\cap} S^*| > |S \widetilde{\cap} S^*|$.*

Conte & Uno [26] proved that, if the problem of enumerating all maximal $\mathcal{P}$-subsets of E is proximity searchable, then Algorithm 2.4 generates each solution exactly once and runs with polynomial delay using exponential space.

Note that being able to find one first solution in polynomial time is necessary to get a polynomial delay algorithm. One also needs to be able to check if a subset is a solution in polynomial time. Moreover, if exponential space is allowed, it is possible to test in time linear in $|E|$ if a solution S' has already been found, using a dictionary in which insertion and verification can be done in time polynomial in the size of S'.

[1] In mathematics, a function $d : X \times X \to \mathbb{R}_{\geqslant 0}$ on a set X is called a *distance* if it satisfies the properties of symmetry ($\forall x, y \in X,\ d(x, y) = d(y, x)$), separation ($\forall x \in X,\ d(x, x) = 0$), and the triangle inequality ($\forall x, y, z \in X,\ d(x, y) + d(y, z) \leqslant d(x, z)$).

Algorithm 2.4: The general Proximity Search algorithm

input : A ground set E, a property $\mathcal{P}$, with a polynomially computable function NEIGHBOURS
output : All maximal subsets of E satisfying $\mathcal{P}$

$\mathcal{S} := \varnothing$;
$S :=$ an arbitrary maximal subset of E satisfying $\mathcal{P}$;
Run ENUM$(S, \mathcal{S})$;

Function ENUM $(S, \mathcal{S})$**:**
    $\mathcal{S} := \mathcal{S} \cup \{S\}$;
    Output S if recursion depth is even;
    foreach $S' \in$ NEIGHBOURS(S) **do**
        if $S' \notin \mathcal{S}$, then ENUM$(S', \mathcal{S})$;
    Output S if recursion depth is odd;

Proximity Search has been proved efficient for many maximal subgraphs enumeration problems. For example, it can be used to enumerate all maximal (induced or not) bipartite subgraphs, chordal subgraphs, k-degenerate subgraphs, maximal connected obstacle-free convex hulls, with polynomial delay [26]. We will show in detail several more applications of the Proximity Search framework in Chapter 5.

The Proximity Search framework is well-suited for enumerating maximal subgraphs, coping with some of the limitations of the Input Restricted Problem approach, but it usually requires exponential space. Refinements have been proposed [24, 23], under certain conditions on the set system, in particular to make Algorithm 2.4 run in polynomial space. In Chapter 6, we will propose a new refinement of Proximity Search with weaker assumptions that achieves polynomial space as well.

2.2.6 Cao's retaliation-free paths

Recently, in a preprint of 2020 [13], Cao came up with a new technique for the enumeration of maximal induced subgraphs. This technique, called *retaliation-free paths*, appears to make the enumeration of maximal induced subgraphs possible for several more classes of graphs. It is namely used to enumerate the maximal induced trivially perfect subgraphs of a given graph.

The idea behind this technique is the same as in Proximity Search: it consists in showing the strong connectivity of a supergraph of solutions. First, one has to define a suitable, problem-specific neighbouring function, as in Proximity Search. However, Cao manages to build paths in which the usual proximity measure is not increasing. Indeed, the important property of retaliation-free paths is that any time an element belonging to the target solution is removed, it is possible to make sure it will be re-introduced

at a later step. This way, he asserts that, if the problem satisfies certain conditions and the neighbouring function is correctly defined, there always exists a retaliation-free path from one solution to another in the supergraph of solutions. Then, a depth-first search on the space of solutions allows to enumerate all of them with polynomial delay.

2.3 Enumeration of maximal subgraphs and minimal supergraphs

The problems on which we focus here are the enumeration of maximal induced $\mathcal{P}$-subgraphs, minimal $\mathcal{P}$-deletions and minimal $\mathcal{P}$-completions of an arbitrary graph G, for an *hereditary* graph property $\mathcal{P}$. These problems are defined in Chapter 1.

In some cases it is not possible, let alone meaningful, to find the optimal solution to such a problem. Indeed, the notion of "best" solution can be hard to model in terms of graphs, and in particular obtaining the minimum or maximum cardinality solution can be irrelevant to the problem. It is for example the case when dealing with databases [18], bioinformatics [12], or statistical models [27]. Therefore, it is of better use to be able to enumerate all the maximal, or minimal, solutions: after that, the user can choose themself which solution is the best according to their specific criteria.

When the corresponding optimisation problem is hard, enumeration can also be used for sampling purposes. Since the computation can be stopped at any time, one can choose the best solution among those generated up to a certain time.

All the properties we consider here must be recognisable in polynomial time. Indeed, it can happen that the input graph G already satisfies property $\mathcal{P}$. In this case, the minimal $\mathcal{P}$-completions (resp. minimal $\mathcal{P}$-deletions, maximal induced $\mathcal{P}$-subgraphs) enumeration problem has only one solution, that is G itself: in other words, enumeration generalises recognition. Thus, an output-polynomial algorithm solving our enumeration problem must run in polynomial time when called on input G. This implies that one must be able to determine if G fulfils $\mathcal{P}$ in polynomial time, as stated in the following theorem. That is why in the present work, we are only interested in polynomial time recognisable graph properties.

Theorem 2.6 (Folklore)**.** *If all maximal induced $\mathcal{P}$-subgraphs (resp. minimal $\mathcal{P}$-completions, minimal $\mathcal{P}$-deletions) can be enumerated in output-polynomial time, then the property $\mathcal{P}$ can be recognised in polynomial time.*

From an algorithmic point of view, the problems of enumerating maximal induced subgraphs, minimal deletions or minimal completions are closely linked to the classical enumeration problems presented in Section 2.1.3. In the present section, we will see how those problems are related.

Since the property under consideration is hereditary, the set of all induced $\mathcal{P}$-subgraphs of G is an independent set system. Therefore, we know that in some cases it is possible to enumerate all maximal induced $\mathcal{P}$-subgraphs of G with the output-polynomial algorithm designed by Lawler *et al.* [62] (see Section 2.2.2). But better output-sensitive complexity can be achieved.

Given a graph G and a hereditary property $\mathcal{P}$, one can build an hypergraph on $V(G)$ whose hyperedges are the sets of vertices inducing forbidden induced subgraphs for $\mathcal{P}$ in G. Then, enumerating all maximal induced $\mathcal{P}$-subgraphs of G is equivalent to enumerating all minimal transversals of this hypergraph. Indeed, a minimal transversal of the hypergraph formed by the forbidden induced subgraphs corresponds to a minimal set of vertices of which the removal from the original graph yields a maximal induced subgraph satisfying the property $\mathcal{P}$. Such an hypergraph is represented in Figure 2.7, when the considered property $\mathcal{P}$ refers to "belonging to the class of cographs", that is, avoiding the four-vertex path P_4 as an induced subgraph.

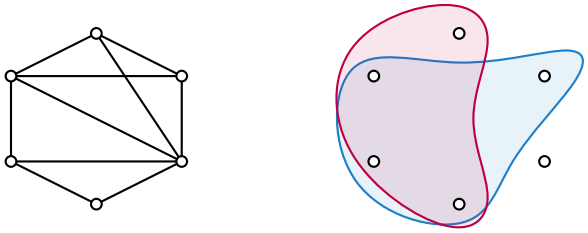

Figure 2.7: From this graph that is not a cograph, one can build an hypergraph whose hyperedges are exactly the sets of vertices inducing a P_4 in the graph.

It is worth noticing that some hereditary graph properties (not all of them) admit a finite number of forbidden induced subgraphs: this is for example the case with split graphs, cographs, and threshold graphs, but not with chordal graphs. When considering such a property, our problem falls into a special case of the general transversals enumeration problem, where the size of the hyperedges is bounded by a constant.

In the 1990s, Eiter & Gottlob worked on this variant of the general problem and exhibited an incremental-polynomial algorithm for the enumeration of minimal transversals of hypergraphs of bounded edge size [41]. Thanks to their result, it is now straightforward to obtain an incremental-polynomial algorithm for the maximal induced $\mathcal{P}$-subgraphs enumeration problem.

If in addition the considered property is monotone, *i.e.* closed under edge removal, then the maximal (non-induced) $\mathcal{P}$-subgraphs, or minimal $\mathcal{P}$-deletions, of G can also be enumerated with Eiter & Gottlob's incremental-polynomial algorithm [41], by considering an hypergraph whose vertices are the edges of G.

As for minimal $\mathcal{P}$-completions of G, provided that $\mathcal{P}$ is a monotone property, they form a dependent set system of which we aim to enumerate the inclusion-wise minimal elements. Plus, they correspond to minimal deletions of $\overline{G}$ in the complement class

of $\mathcal{P}$. In other words, if $\overline{\mathcal{P}}$ denotes the graph property "the complement graph fulfils property $\mathcal{P}$", then up to taking the complement, minimal $\mathcal{P}$-completions of G are exactly minimal $\overline{\mathcal{P}}$-deletions of $\overline{G}$. This allows us to use the same technique as for minimal $\mathcal{P}$-deletions to obtain an incremental-polynomial enumeration algorithm.

At the moment, we know nothing about a general algorithm that could enumerate maximal (induced) $\mathcal{P}$-subgraphs or minimal $\mathcal{P}$-completions of a graph G with polynomial delay for any hereditary graph property $\mathcal{P}$. On the other hand, neither do we know a hereditary graph property $\mathcal{P}$ for which the enumeration of maximal (induced) $\mathcal{P}$-subgraphs or minimal $\mathcal{P}$-completions of a graph G *cannot* be enumerated with polynomial delay. This raises some interesting questions which, although motivating our research, remain open at the end of this thesis.

Question 1. *Is there a (hereditary) graph property $\mathcal{P}$ such that minimal $\mathcal{P}$-completions, or minimal $\mathcal{P}$-deletions,* cannot *be enumerated with polynomial delay?*

Question 2. *More generally, is there a (hereditary) graph property $\mathcal{P}$ such that minimal $\mathcal{P}$-completions, or minimal $\mathcal{P}$-deletions,* cannot *be enumerated in output-polynomial time?*

Chapter 3

Split graphs

We provide in the present chapter efficient algorithms for the enumeration of minimal split completions, deletions, and maximal induced split subgraphs. A part of these results (minimal split completions and deletions) appears as a joint work with Aurélie Lagoutte, Vincent Limouzy, Arnaud Mary and Lucas Pastor in the 2020 preprint Efficient enumeration of maximal split subgraphs and sub-cographs and related classes *[10], submitted.*

In this chapter, we will see an example of how, for a given graph property, the structure can be used to reduce an enumeration problem to another. In particular, the property considered here corresponds to "belonging to the class of *split graphs*".

More precisely, we show how to link the minimal split completions to the maximal stable sets of an auxiliary graph that can easily be found from the input graph. Split graphs have been studied in detail in order to find one single minimal split completion of a given graph [55].

As for maximal induced subgraphs, this class is a good example where the Input Restricted Problem can be used to achieve polynomial delay and polynomial space.

3.1 Minimal split completions

In this section, our aim is to enumerate all minimal split completions of G into a split graph, using a polynomial delay polynomial space algorithm. To do so, we establish a link between the minimal split completions of a graph and its maximal stable sets. However, we will see that different maximal stables sets may lead to the same completion. To overcome this issue, we first perform a pre-processing step that

removes some unnecessary or undesired vertices from the input graph. Once the process is completed, the bijection between both objects is established.

Recall that a minimal split completion of a graph G is an inclusion-wise minimal split supergraph of G on the same vertex set; that is to say a split graph H such that $V(H) = V(G)$, $E(G) \subseteq E(H)$, and there exists no split graph $H' = (V(G), E(H'))$ such that $E(G) \subseteq E(H') \subsetneq E(H)$. The edges in $E(H) \setminus E(G)$ are the fill edges of the split completion. A graph can be seen in Figure 3.1. It is not split, but the addition of three red edges turns it into a split graph: it is a split completion of the input graph.

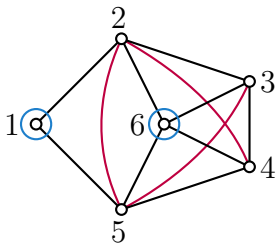

Figure 3.1: The graph with the black edges is not split, but adding the three red (curved) edges produces a split completion of it. The two circled vertices form an inclusion-wise maximal stable set.

Let us begin with a simple observation. Since the class of split graphs is self-complementary, we can list all maximal split deletions of G simply by computing all minimal split completions of the complement graph $\overline{G}$, and then complementing the solutions. Therefore, the same complexity results will be obtained for minimal split deletions as well.

First, note that a split completion of G can always be obtained from a stable set. Indeed, if S is a stable set of G, adding all possible edges between pairs of vertices of $V \setminus S$ makes $V \setminus S$ into a clique, and the resulting graph is a split completion H of G. We will say that H is the split completion *induced* by S. In the example of Figure 3.1, the split completion is induced by the stable set $\{1, 6\}$. In the sequel we will see how to characterise the stable sets producing minimal split completions.

Lemma 3.1. *Let $G = (V, E)$ be a graph. For any minimal split completion H of G, there exists a split partition $V = K + S$ of H such that S is a maximal stable set of G.*

Proof. Let H be a minimal split completion of G. The graph H is split so there exists a split partition, and $H = (S + K, E + F)$ (where F is the set of fill edges). If S is not a maximal stable set of H, there exists $x \in V$ such that $S \cup \{x\}$ is a stable set of H, and of course $K \setminus \{x\}$ is still a clique of H. Consequently, suppose that S is a maximal stable set of H.

For contradiction, suppose that S is not a maximal stable set of G. There exists y such that $S \cup \{y\}$ is a stable set of G. As $S \subseteq S \cup \{y\}$, the split completion induced by $S \cup \{y\}$ is a split sub-completion of H, supposed to be minimal. Therefore $S \cup \{y\}$ is a stable set of H, that is, S is not a maximal stable set of H: contradiction. □

Hence it is only necessary to consider (inclusion-wise) *maximal* stable sets when looking for *minimal* split completions. But this condition is not sufficient. Indeed, in the example of Figure 3.1, the edge $\{3, 5\}$ can be removed and the resulting graph is still a split completion (with split partition $\{2, 4, 5, 6\} + \{1, 3\}$), visible in Figure 3.2. It shows that the split completion induced by the maximal stable set $\{1, 6\}$ is not minimal. Therefore, there exist maximal stable sets that do not induce minimal split completions. This is why we need to push our investigation further and find out which conditions make a maximal stable set induce a minimal split completion.

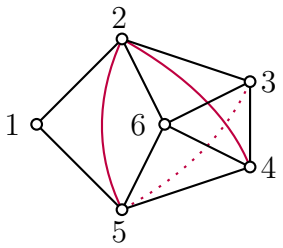

Figure 3.2: Removing the edge $\{3, 5\}$ from the split completion of Figure 3.1 does not break the split property.

In the following, we identify two kinds of vertices which can cause a solution to be non-minimal or produced twice. They are true twins and *redundant vertices*, the definition of which relies on nested neighbourhoods.

Definition 3.2 (Redundant vertices)**.** *Let $G = (V, E)$ be a graph. A vertex $v \in V$ is* redundant *if there exists $u \in V$ such that $N[u] \subsetneq N[v]$.*

Remark that, because of the strict inclusion between the closed neighbourhoods, for every redundant vertex v there exists a vertex u which is not redundant such that $N[u] \subsetneq N[v]$. The set $N[u]$ is inclusion-wise minimal. The interest of identifying redundant vertices is given by the following observation (a stronger version of this will be proved in Lemma 3.3): if S is a maximal stable set of G not inducing a minimal split completion, then S contains a redundant vertex. By the way, the stable set identified in Figure 3.1 is a good example of this: it contains the vertex 6, which is redundant since $N[3] \subsetneq N[6]$.

However, the converse is not necessarily true, as shown on Figure 3.3: in this graph, 2, 4, and 5 are redundant vertices. But there is in this graph a maximal stable set $S = \{2, 5\}$ containing redundant vertices and still inducing a minimal split completion.

Hence, we need to dig deeper into the subject to identify which redundant vertices should be avoided in a stable set S in order for the induced completion to be minimal.

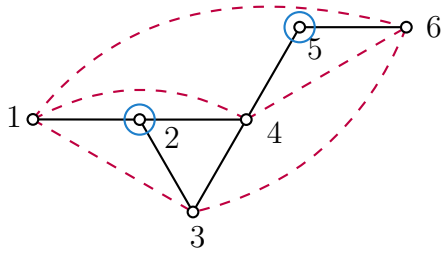

Figure 3.3: In this graph, the vertices 2 and 5 are redundant but the completion induced by the stable set $\{2, 5\}$ (dashed edges) is minimal.

We will show that the right characterisation of minimal split completions is in fact the following.

Lemma 3.3. *Let G be a graph, let I be a maximal stable set of G. The split completion of G induced by I is minimal if and only if I does not contain any redundant vertex x such that $V \setminus N(x)$ is a stable set of G.*

In order to prove Lemma 3.3, we need the following result which establishes a link between the split partitions of a graph and those of its subgraphs. In particular it can be used to compare two different split partitions of a split graph. This lemma will be invoked when characterising non-minimal split completions.

Lemma 3.4 ([55, Observation 4] & Corollary)**.** *Let $G = (V, E)$ and $G' = (V, E')$ be two split graphs such that $E \subseteq E'$, and let $V = I + K$ and $V = I' + K'$ be split partitions of G and G' respectively. Then the following two inequalities hold:*

- $|K' \cap K| \geqslant |K| - 1$;
- $|I' \cup I| \leqslant |I| + 1$.

First, we will try to determine which maximal stable sets are "bad" in the sense that they fail to produce minimal split completions. To this effect, let us find a necessary condition for a maximal stable set to induce a non-minimal split completion. This condition is obtained by looking carefully at the redundant vertices belonging to the considered maximal stable set. We state it as the following lemma.

Lemma 3.5. *Let $G = (V, E)$ be a graph. Let I be a maximal stable set of G. If the split completion of G induced by I is not minimal, there exists a redundant vertex $w \in I$ such that $N(w) = V \setminus I$.*

Proof. Let I be a maximal stable set of G, let $H = (I + C, E + F)$ be the split completion of G induced by I. Assume that the completion H is not minimal. Then there exists a non-empty set of edges $A \subseteq F$ such that $H - A$ is a minimal split completion of G. By Lemma 3.1 there exists a maximal stable set S of G such that $H - A = (S + K, E + F - A)$ is the split completion of G induced by S.

We have the inclusions $G \subseteq H - A \subseteq H$. By Lemma 3.4, $|S| \leqslant |S \cup I| \leqslant |S| + 1$. Because S and I are both maximal stable sets of G we cannot have $I \subseteq S$, so necessarily $|S \cup I| = |S| + 1$. Then,

$$|S| + 1 = |S \cup I| = |S| + |I \setminus S|.$$

Therefore $|I \setminus S| = 1$, and we also have $|S \setminus I| > 0$ because S is maximal. Consequently there exist $w \in V$ and $U \subseteq V$ such that $I = T \cup \{w\}$ and $S = T \cup U$, where U and $T = S \cap I$ are disjoint stable sets of G. Figure 3.4 illustrates this situation.

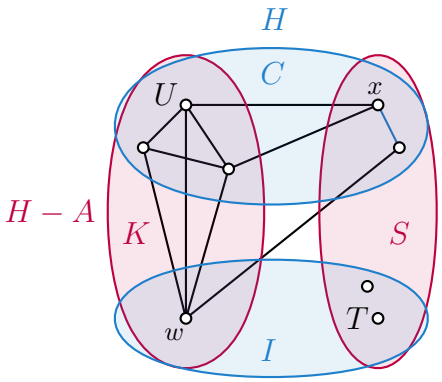

Figure 3.4: A partition of V for the split completion H and another split partition for $H - A$. The vertex w is universal to C. Edges inside $S \cap C$ belong to H but not to $H - A$.

All fill edges of $H - A$ are also fill edges of H. Because w is in I, there is no fill edge incident to w in H, so there is also no fill edge incident to w in $H - A$. As $w \in K$ in $H - A$, this implies $K \cap C \subseteq N(w) \subseteq (K \cap C) \cup U = C$.

Let $x \in U$. As I is a maximal stable set of G, the vertex x has a neighbour in I. This neighbour is not in S because $x \in S$. Therefore, it is in $I \setminus S = \{w\}$, that is, $xw \in E$. Besides, $N[x] \subseteq K \cap C \subseteq N[w]$. This being true for all $x \in U$, we have $N(w) = C = V \setminus I$.

Moreover, the completion H is not minimal, so there exists $y \in U$ incident to a fill edge of H. Hence there exists $z \in C$ such that $yz \notin E$. Then $z \in N[w] \setminus N[y]$, so $N[y] \subsetneq N[w]$ and w is redundant. □

Could it be that the necessary condition of Lemma 3.5 is in fact sufficient? The answer to this question turns out to be positive. We will see that Lemma 3.5 admits a converse, which can be stated as follows.

Lemma 3.6. *Let $G = (V, E)$ be a graph. Suppose that there exist a redundant vertex x such that $V \setminus N(x)$ is a stable set of G. If I is a maximal stable set of G containing x, then the split completion induced by I is not minimal.*

Proof. Let I be a maximal stable set of G containing x. Necessarily $I = V \setminus N(x)$ because $V \setminus N(x)$ is a stable set.

Let $H := (I + V \setminus I, E + F)$ be the split completion induced by I. Since x is redundant, there exists $y \in V$ such that $N[y] \subsetneq N[x]$. The vertex y is a neighbour of x so $y \in V \setminus I$. Therefore y is made adjacent to every vertex of $V \setminus I$ in H. But $N[y] \subsetneq N[x]$, so there exists $z \in V \setminus I$ such that yz is a fill edge in H, as illustrated in Figure 3.5.

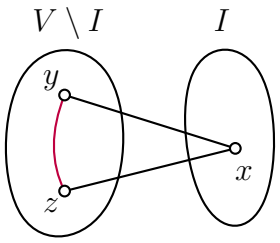

Figure 3.5: There exists z such that $xz \in E$ and yz is a fill edge in H.

The idea is then to move x to the stable set, and y to the clique by considering the graph $H' = H - \{yz \in F \mid z \in N(x)\}$. We observe that H' is a proper sub-split completion of H. Hence H is not minimal. □

From the last two lemmas, it is now straightforward to deduce the exact characterisation announced in Lemma 3.3.

Proof of Lemma 3.3. Lemma 3.5 gives one direction of implication, and Lemma 3.6 gives the other (by noticing that $N(w) = V \setminus I$ rewrites to $I = V \setminus N(w)$). □

In the light of Lemma 3.3, the idea is now to remove from G the identified "bad" redundant vertices, therefore building an auxiliary graph containing only the vertices that for sure will not take part in inducing non-minimal split completions. Then, we will only enumerate split partitions induced by maximal stable sets of this auxiliary graph. But in order to do so, we need to ensure that maximal stable sets in this auxiliary graph are also maximal stable sets in G. This is the purpose of the following lemma.

Lemma 3.7. *Let $G = (V, E)$ be a graph and let R be the set of its redundant vertices. For all $R' \subseteq R$, the maximal stable sets of $G \setminus R'$ are maximal stable sets of G.*

Proof. Let $R' \subseteq R$. Recall that $R = \{v \in V \mid \exists u \in V,\ N[u] \subsetneq N[v]\}$. Observe that all maximal stable sets of $G \setminus R'$ are necessarily stable sets of G. We will see that they are maximal.

Let S be a maximal stable set of $G \setminus R'$. For contradiction, assume that S is not a maximal stable set of G. Hence there exists $v \in V$ such that $S \cup \{v\}$ is a stable set of G, and $v \in R'$ because otherwise S would not be maximal in $G \setminus R'$. The vertex v is redundant so there exists $u \in V \setminus R$ such that $N[u] \subsetneq N[v]$. This way, $S \cup \{u\}$ is a stable set of G and a stable set of $G \setminus R'$: contradiction. □

Nevertheless, we are not yet finished. Given a graph G, we know that removing some identified redundant vertices from G provides a graph in which every maximal stable set gives a minimal split completion of G, and doing so we guarantee to actually get every minimal spit completion of G. However, nothing prevents us from finding the same solution several times. This phenomenon is illustrated in Figure 3.6, where the stable sets are the sets of circled vertices, and the completion consists only in adding the middle horizontal edge.

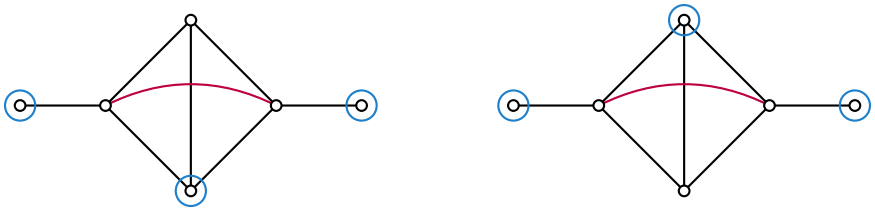

Figure 3.6: The two stable sets represented by circled vertices give the same minimal split completion.

To solve this issue, we will characterise stable sets which give the same minimal completion, among the ones satisfying conditions of Lemma 3.3. This time, the vertices at which we are looking are *true twins*. Remark that in the graph of Figure 3.6, the top vertex and the bottom vertex have the same closed neighbourhood. This leads us to the following result: two distinct maximal stable sets induce the same split completion if and only if their symmetric difference is a pair of true twins.

Lemma 3.8. *Let G be a graph, let S_1 and S_2 be two distinct maximal stable sets of G. Denote by H_1 and H_2 the split completions of G induced by S_1 and S_2 respectively. Define $S := S_1 \cap S_2$. Then S_1 and S_2 induce the same split completion (that is to say, $E(H_1) = E(H_2)$) if and only if there exists a pair of true twins x_1, $x_2 \in V$ such that $S_1 = S \cup \{x_1\}$ and $S_2 = S \cup \{x_2\}$, with $N_G[x_1] = N_G[x_2] = V \setminus S$.*

Proof. Let $K_1 := V \setminus S_1$ and $K_2 := V \setminus S_2$ be cliques in H_1 and H_2, respectively.

Assume S_1 and S_2 induce the same split completion, that is to say $E(H_1) = E(H_2)$. By Lemma 3.4, we have $|S_1 \cup S_2| = |S_1|+1$ and $|S_1 \cup S_2| = |S_2|+1$. Then $|S_1| = |S_2|$ and there exists $x_1 \in S_1 \setminus S_2$ and $x_2 \in S_2 \setminus S_1$ such that $S_1 = S \cup \{x_1\}$ and $S_2 = S \cup \{x_2\}$.

There is no fill edge incident to x_1 in H_1. But because $x_1 \in K_2$, we have $K_1 \cap K_2 \subseteq N_G(x_1) \subseteq (K_1 \cap K_2) \cup \{x_2\}$. Moreover $x_1x_2 \in E(G)$, otherwise $S_1 \cup S_2$ would be a

stable set of G containing S_1, assumed to be maximal. This implies that $N[x_1] = V \setminus S$. By symmetry of the roles of x_1 and x_2, it gives $N[x_1] = N[x_2] = V \setminus S$, and in particular x_1 and x_2 are true twins.

Conversely, if $N[x_1] = N[x_2] = V \setminus S$, there is no fill edge incident to x_1 in H_2, and there is no fill edge incident to x_2 in H_1. Consequently, the only fill edges in H_2 and in H_1 are between vertices of $V \setminus S = K_1 \cap K_2$. Since H_1 and H_2 are both split, they have the same set of fill edges, so $E(H_1) = E(H_2)$. □

Inspired by Lemmas 3.3 and 3.8, we can then construct from every graph G an auxiliary graph $f(G)$. Let $B = \{x \in V(G) \mid x \text{ is redundant and } V \setminus N(x) \text{ is a stable set}\}$. First remove B from G, and observe that $G[V \setminus B]$ has no extra pair of true twins compared to G itself. Then from $G[V \setminus B]$, remove all but one vertex from each set of pairwise true twins, resulting in the graph $f(G)$ having no pair of true twins. The auxiliary graph $f(G)$ is not unique because of the choice between each set of true twins, but is does not matter for the following.

Theorem 3.9. *Let G be a graph. There exists a bijection between the set of all minimal split completions of G and the set of all maximal stable sets of any auxiliary graph $f(G)$.*

Proof. Let B as in the definition of $f(G)$. Observe that $G[V \setminus B]$ cannot contain a redundant vertex x of which the non-neighbourhood in $G[V \setminus B]$ is a stable set, and the same applies to $f(G)$. Moreover, a set $I \subseteq V(f(G))$ is a maximal stable set of $f(G)$ if and only if it is a maximal stable set of $G[V \setminus B]$ included in $V(f(G))$ (because adding a true twin to an existing vertex in $f(G)$ preserves the maximality of I). So, by combining Lemmas 3.7 and 3.3, a set $I \subseteq V(f(G))$ is a maximal stable set of $f(G)$ if and only if the split completion induced by I is a minimal split completion of G. This, combined with the non-existence of true twins in $f(G)$ and Lemma 3.8, ensures the bijection. □

Procedure 3.1 produces the auxiliary graph $f(G)$ mentioned in Theorem 3.9, containing no redundant or true twin vertex v such that $V \setminus N(v)$ is a stable set. During Step 5, every redundant vertex will be marked, as well as all but one vertex of each set of pairwise true twin vertices.

Theorem 3.9 ensures that we can enumerate exactly once every minimal split completion of an input graph G by running Procedure 3.1 followed by an algorithm that enumerates all maximal stable sets of $f(G)$. Since Procedure 3.1 is polynomial in time and space, the complexity of this algorithm only depends on the complexity of the algorithm used for the enumeration of maximal stable sets. We are thus looking for such an algorithm running in polynomial delay and polynomial space. For example, the algorithm proposed by Tsukiyama *et al.* in 1977 [84] fulfils these complexity requirements and allows us to state the following.

Procedure 3.1: CONSTRUCTION OF THE AUXILIARY GRAPH

input : a graph G

```
/* Mark redundant and twin vertices */;
Begin with all vertices unmarked;
foreach unmarked x ∈ V do
    foreach unmarked y ∈ N(x) do
        if N[x] ⊆ N[y] then
            mark y;
/* Remove unnecessary vertices */;
foreach marked x ∈ V do
    if V \ N_G[x] is a stable set then
        remove x;
```

Theorem 3.10. *Minimal split completions (resp. deletions) of a graph can be enumerated with polynomial delay in polynomial space.*

According to Lemma 3.1, the number of minimal split completions of a graph G is at most the number of its maximal stable sets. It has been proven by Moon & Moser in 1965 [66] that this number can be as large as $3^{n/3}$, reached for a graph composed of $n/3$ disjoint triangles as in Figure 3.7.

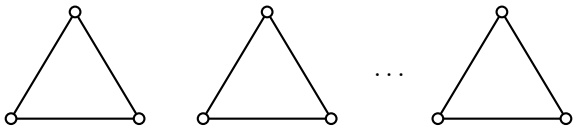

Figure 3.7: A graph built from $n/3$ disjoint triangles.

Lemma 3.3 states that there is a bijection between the minimal split completions and the maximal stable sets of a graph G with no redundant vertex u such that $V(G) \setminus N(u)$ is a stable set. The graph presented in Figure 3.7 verifies this property. In particular, it has $3^{n/3}$ minimal split completions. From these observations we deduce the following theorem.

Theorem 3.11. *The upper bound $3^{n/3}$ on the number of minimal split completions is tight.*

By adding a universal vertex to the graph of Figure 3.7, we obtain a graph with one redundant vertex (the universal vertex). The number of its minimal split completions is therefore equal to the number of maximal stable sets of the graph shown in Figure 3.7,

that is, $3^{(|V(G)|-1)/3}$. Therefore, we can even consider our graph to be connected and still have $\mathcal{O}(3^{n/3})$ minimal split completions.

Using a polynomial delay algorithm for the enumeration of the maximal stable sets of a graph, our algorithm for the enumeration of all minimal split completions runs in total time $\mathcal{O}(Npoly(n))$, where N is the number of maximal stable sets. Therefore, its total execution time is bounded in an input-sensitive manner by $\mathcal{O}(3^{n/3}poly(n))$, almost matching the bound of Theorem 3.10.

Corollary 3.12. *There exists a polynomial delay algorithm for the enumeration of all minimal split completions of a graph running in total time* $\mathcal{O}(3^{n/3}poly(n))$.

3.2 Maximal induced split subgraphs

As it was proved in the previous section, we are able to enumerate efficiently all minimal split completions and deletions of an input graph G. Now, can similar complexity results be achieved for the enumeration of maximal induced subgraphs? We addressed this question in 2019, that was independently solved by Cao in a preprint of 2020 [13] with a similar technique: the input restricted problem is polynomial, therefore it can be used to compute all maximal induced split subgraphs of G with polynomial delay and polynomial space. For the sake of completeness, our results, left unpublished, are presented in this section.

Let us begin with an observation. For $G = (V, E)$ a graph, an induced split subgraph of G can be constructed by considering a clique K of G and a stable set S of $G \setminus K$. The graph $G[K \cup S]$ is then a split subgraph of G with split partition $K + S$. The next theorem is here to specify which cliques and stable sets have to be considered when looking for maximal induced split subgraphs.

Theorem 3.13. *Let G be a graph, and let H be a maximal induced split subgraph of G. There exists K a maximal clique of G and S a maximal stable set of $G \setminus K$ such that $H = G[K \cup S]$.*

Proof. Since H is split, it admits a split partition $C + I$ where C is a clique and I is a stable set. Assume that C is not a maximal clique in G. By definition there exists an $x \in V \setminus C$ such that $C \cup \{x\}$ is a clique in G.

If $x \notin I$, then $G[C \cup I \cup \{x\}]$ is an induced split subgraph of G that strictly contains H. This cannot happen since we assumed H maximal. Therefore $x \in I$. In this case, we can set $K = C \cup \{x\}$ and $S = I \setminus \{x\}$. Now, any $y \in S$ is non-adjacent to x, thus K is a maximal clique of G.

Moreover, S is a stable set of $G \setminus K$. If S were not maximal, there would exist S_0 a stable set of $G \setminus K$ such that $S \subsetneq S_0$, and H would be an induced split subgraph of the split graph $G[K \cup S_0]$, which is excluded by maximality of H.

Hence $H = G[K \cup S]$ with K a maximal clique of G and S a maximal stable set of $G \setminus K$. □

A maximal split subgraph can therefore be obtained by choosing a maximal clique, and a maximal stable set of the rest of the graph. This construction is illustrated in Figure 3.8.

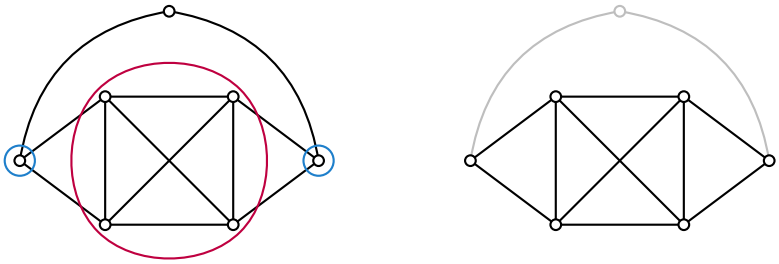

Figure 3.8: The maximal induced split subgraph on the right was obtained from the maximal clique and stable set identified in the graph on the left.

The idea now is to prove that the input restricted problem for maximal induced split subgraphs is polynomial. To this effect, consider a maximal induced split subgraph H of the input graph G, and a vertex $v \in V(G) \setminus V(H)$. Then, we upper-bound the number of couples (K, S) such that K is a maximal clique of $H + v$ and S is a maximal stable set of $(H + v) \setminus K$, and show that their number is at most quadratic. Finally, it suffices to enumerate all such couples (K, S) with an output-polynomial algorithm: since the number of solutions is polynomial in the size of $H + v$, all of them will be generated in polynomial time.

Let us have a look on what happens when we add a vertex to a maximal induced split subgraph H of G. The following lemma links the neighbourhood of the new vertex to the split partitions of H.

Lemma 3.14. *Let H be a maximal induced split subgraph of G, and let v be a vertex in $V(G) \setminus V(H)$. Denote by (K, S) a split partition of H. Then $K \setminus N(v) \neq \varnothing$, and $S \cap N(v) \neq \varnothing$.*

Proof. Assume for contradiction that $K \setminus N(v) = \varnothing$, that is, $K \subseteq N(v)$. Then $K \cup \{v\}$ is a clique in $H + v$. Since S is a stable set in H, then the vertices of $H + v$ can be partitioned in two sets $K \cup \{v\}$ and S, where $K \cup \{v\}$ is a clique and S is a stable set. Therefore, $H + v$ is an induced split subgraph of G, contradicting the maximality of H.

Now, if $S \cap N(v) = \varnothing$, then $S \cup \{v\}$ is a stable set in $H + v$. As before, we deduce that $H + v$ has a split partition $(K, S \cup \{v\})$, and $H + v$ is an induced split subgraph of G, contradicting the maximality of H. □

The situation described in Lemma 3.14 can be represented as in Figure 3.9, where v is the added vertex. As stated previously, v has at least one neighbour in the stable set and one non-neighbour in the clique.

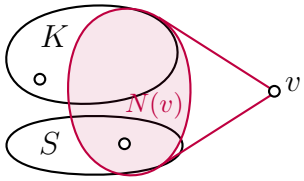

Figure 3.9: A graph that is not split but can be turned into a split graph with split partition $K + S$ by removing the vertex v.

The next lemma bounds the number of maximal cliques in $H + v$.

Lemma 3.15. *Let H be a maximal induced split subgraph of G, and let v be a vertex in $V(G) \setminus V(H)$. Let $n = |V(H) \cup \{v\}|$ be the number of vertices of $H + v$. Then $H + v$ has at most $2n$ maximal cliques.*

Proof. The graph H is split: let (K, S) be any split partition of the vertices of H. By Lemma 3.14, there are exactly two maximal cliques of $H + v$ which do not contain vertices of S: these are K and $(K \cap N(v)) \cup \{v\}$. Then, for any vertex $u \in S$, the only maximal cliques of $H+v$ that contain u are $(N(u) \setminus \{v\}) \cup \{u\}$, and $(N(u) \cap N(v)) \cup \{u, v\}$ if $uv \in E$. Therefore, there are at most $2n$ maximal cliques in $H + v$. □

Lemma 3.15 directly implies the following corollary, upper-bounding the number of couples (K, S) where K is a maximal clique of $H + v$ and S is a maximal stable set of $(H + v) \setminus K$.

Corollary 3.16. *Let H be a maximal induced split subgraph of G, and let v be a vertex in $V(G) \setminus V(H)$. There exists at most a quadratic number of couples (K, S) such that K is a maximal clique of $H + v$ and S is a maximal stable set of $(H + v) \setminus K$.*

Proof. By Lemma 3.15, there are at most a linear number of maximal cliques in $H + v$. Let K be any maximal clique of $H + v$.

If $v \in K$, then $(H + v) \setminus K$ is split (as an induced subgraph of H). Consequently, it contains a linear number of stable sets.

Otherwise, $(H + v) \setminus K$ can be written as $H' + v$ with H' an induced subgraph of H. Since H is split, so is H'. Therefore, the complement of $H' + v$ meets the requirements of Lemma 3.15: it contains a linear number of maximal cliques.

In other words, $(H + v) \setminus K$ contains a linear number of maximal stable sets for any maximal clique K of $H + v$. This concludes the proof. □

Now, it remains to generate all such couples (K, S) in polynomial time. It can be achieved with two nested calls to the algorithm of Tsukiyama *et al.* [84], that was already used for minimal split completions. This algorithm even runs in polynomial space. This is why in our case, the function genRestrHered used in Algorithm 2.3 can be computed in polynomial time and polynomial space.

Therefore, it is possible to use Algorithm 2.3 to enumerate all maximal induced split subgraphs of a graph G with polynomial delay. Due to the space complexity of the algorithm used to solve the input restricted problem, polynomial space can also be achieved. The result is summarised here as Theorem 3.17.

Theorem 3.17. *The maximal induced split subgraphs of an arbitrary input graph can be enumerated with polynomial delay in polynomial space.*

Chapter 4

Extension problems

In this chapter we investigate the Flashlight Search technique, presented in Chapter 2, and prove some hardness results on the so-called extension problem*, which prevent from finding a general enumeration algorithm based on Flashlight Search. These results appear as a joint work with Aurélie Lagoutte, Vincent Limouzy, Arnaud Mary and Lucas Pastor in the 2020 preprint* Efficient enumeration of maximal split subgraphs and sub-cographs and related classes *[10].*

An interesting result would be to find a general method that, given a (hereditary) graph property $\mathcal{P}$ and an input graph G, returns all maximal induced $\mathcal{P}$-subgraphs of G. Such a general method would also be of great interest for the minimal $\mathcal{P}$-completion and $\mathcal{P}$-deletion problems. Because of its generality, Flashlight Search (see Section 2.2.1 for an introduction) could seem appropriate for this purpose. Moreover, this method has been successfully used on several special cases [25, 60], when the property $\mathcal{P}$ is restricted, or if the graph G is constrained to a particular graph class.

We will show in this chapter that unfortunately, Flashlight Search needs some restrictions on the input: it cannot in general return all maximal induced $\mathcal{P}$-subgraphs of G for any $\mathcal{P}$ and any G with polynomial delay and polynomial space. Moreover, even if it can provide a polynomial delay and polynomial space algorithm for the enumeration of minimal $\mathcal{P}$-deletions for some hereditary properties, we show that there are other graph properties for which Flashlight Search is not the best tool.

More precisely, we define the *extension problem* (for induced subgraphs and for deletions) as the problem that must be solved in polynomial time in order to use Flashlight Search, and we show that this problem is NP-complete in many cases.

4.1 Maximal induced subgraphs

Extension problem. For a hereditary graph property $\mathcal{P}$, the *extension problem for maximal induced $\mathcal{P}$-subgraphs* is defined as follows. Given a graph G (not satisfying $\mathcal{P}$ in general) and two disjoint subsets A and B of its vertex set V, decide if there exists a maximal induced subgraph H of G satisfying $\mathcal{P}$, containing A and avoiding B.

In other words, we want that $A \subseteq V(H)$ and $B \cap V(H) = \varnothing$. Note that we cannot remove B from the vertex set before looking for a maximal induced $\mathcal{P}$-subgraph: this operation could lead us to a non-maximal induced $\mathcal{P}$-subgraph of G, which is not what we are looking for.

If it were possible to solve the extension problem for $\mathcal{P}$ in polynomial time, then we would be able to enumerate all maximal induced $\mathcal{P}$-subgraphs of a given graph G using Flashlight Search. Being able to decide in polynomial time if there exists a maximal induced $\mathcal{P}$-subgraph of G containing a set of prescribed vertices and avoiding another is exactly what is needed at each step of the algorithm. Indeed, at step i of the binary partition, the two sets of vertices are included in $\{e_1, \ldots, e_i\}$, and we have to decide whether or not there exists a compatible solution.

Recall that a graph property $\mathcal{P}$ is hereditary if for all G satisfying $\mathcal{P}$, all induced subgraphs of G also satisfy $\mathcal{P}$. The hereditary property $\mathcal{P}$ is called *non-trivial* if it is verified by infinitely many graphs and has at least one obstruction, which is a graph $J_{\mathcal{P}}$ not satisfying $\mathcal{P}$. In other words, a hereditary graph property is non-trivial if infinitely many graphs satisfy it, and infinitely many do not.

The *node-deletion problem* consists in determining the *minimum* number of nodes which must be deleted from a graph such that the resulting subgraph satisfies the property $\mathcal{P}$. It has been shown by Lewis & Yannakakis in 1980 [63] that for any non-trivial hereditary property $\mathcal{P}$, the node-deletion problem is NP-complete, provided that testing for $\mathcal{P}$ can be performed in polynomial time. We will show here that the same can be said about the extension problem.

Theorem 4.1. *For any non-trivial hereditary graph property $\mathcal{P}$, the extension problem for maximal induced $\mathcal{P}$-subgraphs is NP-hard.*

Moreover, if $\mathcal{P}$ can be tested in polynomial time, then the extension problem is NP-complete.

Let us recall that "the class $\mathcal{P}$" denotes in fact the class of all graphs satisfying property $\mathcal{P}$. First of all, let us observe that the class $\mathcal{P}$ contains all the stable sets or all the cliques. Indeed, because $\mathcal{P}$ is non-trivial there exist graphs in $\mathcal{P}$ that have arbitrarily many vertices. Let $k \in \mathbb{N}$. Then by Ramsey's theorem, every graph in $\mathcal{P}$ that is large enough (at least the so-called Ramsey number $R(k, k)$) contains either a clique or a stable set of size k. Since $\mathcal{P}$ is a hereditary property, it implies that this clique or stable set of size k belongs to $\mathcal{P}$. Up to considering the complement class

$\overline{\mathcal{P}}$ (which remains non-trivial and hereditary), we can assume that $\mathcal{P}$ contains all the stable sets.

The first step is to define a preorder (a binary relation that is symmetric and transitive) on the set of all graphs, essentially the same as the one given by Lewis & Yannakakis [63]. This preorder will be useful in the proof of Theorem 4.1, to ensure that the considered graph satisfies property $\mathcal{P}$.

For any connected graph G and any vertex $c \in V(G)$, let $\lambda^c(G)$ be the non-increasing sequence of the sizes of the connected components of $G - c$. If $V(G) = \{c\}$, then $\lambda^c(G) = (0)$. Then, let $\lambda(G)$ be the minimum for the lexicographic order of the $\lambda^c(G)$, that is, $\lambda(G) = \min_{lex} \{\lambda^c(G) \mid c \in V(G)\}$. Let $c(G) \in V(G)$ be the vertex minimising $\lambda^c(G)$. If the minimum is reached for several vertices, take any of them (for example the lexicographically smallest) as $c(G)$. By definition, $\lambda(G) = \lambda^{c(G)}(G)$.

For any graph G with connected components $G_1, \ldots, G_l$, let $\mu(G)$ be the non-increasing sequence of the $\lambda(G_i)$ according to the lexicographic order. For convenience, the connected components $G_1, \ldots, G_l$ of G will now be ordered to have $\mu(G) = (\lambda(G_1), \ldots, \lambda(G_l))$. The function μ defines a prewellordering (all elements are pairwise comparable, there is no infinite decreasing sequence for μ) on the set of all graphs.

Figure 4.1 illustrates this preorder on four connected graphs G_1, G_2, G_3 and G_4. The vertex c minimising λ is identified in G_1 and G_3, whereas for G_2 and G_4, any vertex can be taken. Remark that in the induced preorder, μ can take the same value on several graphs: in this example, $\mu(G_2) = \mu(G_4) = ((4))$.

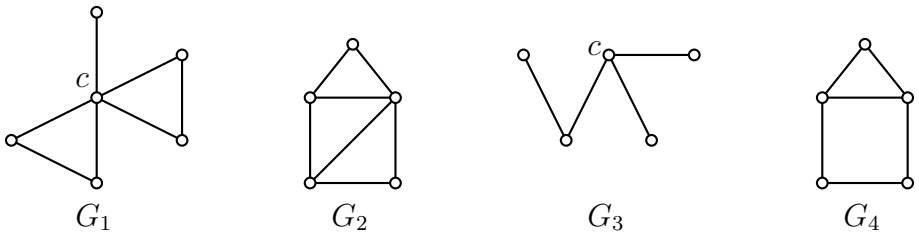

Figure 4.1: In this example, we have $\mu(G_1) = ((2, 2, 1))$ whereas $\mu(G_2) = \mu(G_4) = ((4))$, and $\mu(G_3) = ((2, 1, 1))$. Therefore, $\mu(G_3) \leqslant \mu(G_1) \leqslant \mu(G_2)$, and G_3 is the smallest of the four graphs for μ.

Equipped with this prewellordering on the set of all graphs, we are now ready to prove Theorem 4.1.

Proof of Theorem 4.1. First, if $\mathcal{P}$ can be tested in polynomial time, the extension problem is in NP. Indeed, given a subset S of vertices of the input graph G, one can easily check in polynomial time whether $G[S]$ satisfies $\mathcal{P}$, and whether S contains A

and avoids B. Moreover, the maximality of $G[S]$ can be tested in polynomial time because $\mathcal{P}$ is assumed to be hereditary.

As stated in the previous discussion, it is always possible to assume that the class $\mathcal{P}$ contains all the stable sets.

The preorder defined by μ is well-founded, so every non-empty set of graphs has a smallest element for μ. In particular, the set of all graphs that do *not* satisfy $\mathcal{P}$ has a smallest element for μ. Hence there exists a graph $J_{\mathcal{P}} \notin \mathcal{P}$ which is minimum for μ, that is to say $\mu(J_{\mathcal{P}}) = \min_{lex}\{\mu(H) \mid H \notin \mathcal{P}\}$. Since $\mathcal{P}$ contains all the stable sets, the graph $J_{\mathcal{P}}$ has at least one edge, thus at least two vertices.

Denote by $J_1, \ldots, J_l$ the connected components of $J_{\mathcal{P}}$. Therefore $\mu(J_{\mathcal{P}})$ is the sequence $\mu(J_{\mathcal{P}}) = (\lambda(J_1), \ldots, \lambda(J_l))$. Let J_0^* be the largest connected component of $J_1 - c(J_1)$. We define J_0 as the subgraph of $J_{\mathcal{P}}$ induced by $V(J_0^*) \cup \{c(J_1)\}$. Consider also the induced subgraph $J_1' := J_{\mathcal{P}}[V(J_1) \setminus V(J_0^*)]$. Both graphs J_0 and J_1' are connected, and J_0 has at least one edge. Let then d be any vertex of J_0 different from $c(J_1)$.

Before starting the reduction, let us have a look at what happens to the obstruction $J_{\mathcal{P}}$, and how J_0 and J_1' are built. An illustration of the process applied to a graph $J_{\mathcal{P}}$ is given in Figure 4.2. This graph satisfies $\mu(J_{\mathcal{P}}) = ((4,2),(3))$.

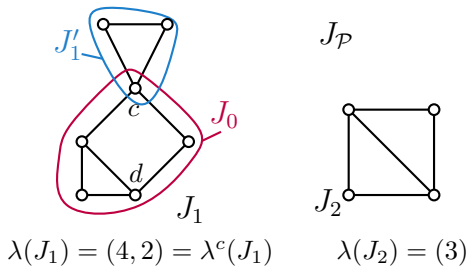

Figure 4.2: An obstruction $J_{\mathcal{P}}$ and its connected components J_1 and J_2. The two subgraphs J_0 and J_1' are highlighted.

The reduction is with the problem of finding a maximal stable set avoiding a set $B \subseteq V(G)$, whose NP-completeness has been proven by Boros *et al.* [7, Proposition 2].

Let G be a graph, and let B be a subset of its vertices. We build a graph G', using a technique similar to the one used for showing that the node-deletion problem is NP-complete [63], as follows.

For each vertex u of G, attach a copy of J_1' to u by identifying $c(J_1)$ with u. Replace every edge of G by a copy of J_0, identifying $c(J_1)$ with one endpoint and d with the other, in any order. Finally, add $J_2, \ldots, J_l$ as new connected components. The graph G' is the graph obtained this way. Figure 4.3 illustrates the transformation of a graph G into G', using the obstruction $J_{\mathcal{P}}$ of Figure 4.2. By construction, the graph G' contains many copies of $J_{\mathcal{P}}$, therefore it does not satisfy the hereditary property $\mathcal{P}$.

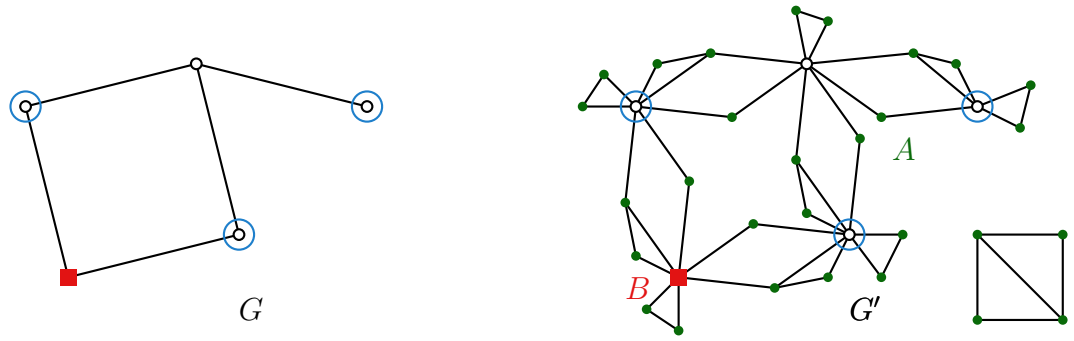

Figure 4.3: G transforms into G'. The set B consists here in the only square vertex, and the set A in the smaller vertices. The circled vertices form a maximal stable set of G not intersecting B.

Call A the set of all vertices that are in G' but were not in G. We will show that finding a maximal stable set of G avoiding B, or finding a maximal induced $\mathcal{P}$-subgraph of G' which contains A and avoids B are two equivalent problems.

1. First, suppose that there exists a maximal stable set S of G which does not intersect B. We will see that the graph $H := G'[S \cup A]$ verifies $\mu(H) < \mu(J_{\mathcal{P}})$.

As S is a maximal stable set of G, each edge in G has at least one endpoint in $V \setminus S$. Hence, removing from G' all the vertices of $V(G) \setminus S$ destroys a vertex identified to $c(J_1)$ or d in each copy of J_1.

For each vertex $s \in S$, let us further observe the connected component H_s of H containing s. It is composed of one copy of J_1' and $d_G(s)$ "truncated" copies of J_0, all glued together by identifying either $c(J_1)$ or d into s (by "truncated" copy of J_0, we mean either $J_0 - c(J_1)$ or the connected component of $J_0 - d$ that contains $c(J_1)$). It follows $\lambda(H_s) \leqslant \lambda^s(H_s) < \lambda(J_1)$ because the truncated copies of J_0 are smaller than J_0 itself.

Now, let us show that $\mu(H) < \mu(J_{\mathcal{P}})$. Observe that the connected components of H are precisely $\{H_s : s \in S\} \cup \{J_2, ..., J_l\}$. We assumed that $\lambda(J_1), ..., \lambda(J_l)$ is a non-increasing sequence, so $\lambda(J_i) \leqslant \lambda(J_1)$ for all $2 \leqslant i \leqslant l$.

If $\lambda(J_2) < \lambda(J_1)$ (if the inequality is strict), then since $\lambda(H_s) < \lambda(J_1)$ for all $s \in S$ we conclude that $\lambda(C) < \lambda(J_1)$ for any connected component C of H. This implies that $\max\{\lambda(C) : C \text{ is a connected component of } H\} < \lambda(J_1)$. Therefore the sequence $\mu(H)$ is lexicographically strictly smaller than $\mu(J_{\mathcal{P}})$ since the first element of $\mu(H)$ is strictly smaller than the first element of $\mu(J_{\mathcal{P}})$.

If $\mu(G)$ was starting with a plateau, that is to say, if there exists $j \geqslant 2$ such that $\lambda(J_1) = \lambda(J_2) = \ldots = \lambda(J_j)$ (choose the largest such index j), then we conclude by noticing that the sequence $\mu(H)$ starts with the plateau $\lambda(J_2) = \ldots = \lambda(J_j)$, which is strictly shorter by one, and then all other values appearing in $\mu(H)$ are strictly

smaller than $\lambda(J_1)$. Indeed, since for all $s \in S$, $\lambda(H_s) < \lambda(J_1)$ and since the only other connected components of H are copies of $J_2, ..., J_l$, the only connected components C of H for which $\lambda(C) = \lambda(J_1)$ are precisely the copies of $J_2, ..., J_j$. So only the first $j-1$ elements of the sequence $\mu(H)$ will have value $\lambda(J_1)$ and then, the sequence $\mu(H)$ will be lexicographically strictly smaller than $\mu(J_{\mathcal{P}})$.

In either case, we obtain that $\mu(H) < \mu(J_{\mathcal{P}})$. As $J_{\mathcal{P}}$ is the smallest obstruction for $\mathcal{P}$, and since by definition of μ, $\mu(H)$ is larger than $\mu(K)$ for any subgraph K of H, the graph H does not contain any obstruction as an induced subgraph and $H \in \mathcal{P}$. Therefore H is an induced $\mathcal{P}$-subgraph of G' that contains A and does not intersect B.

2. Conversely, assume that there exists a maximal induced $\mathcal{P}$-subgraph H of G' containing A and avoiding B. Then $J_{\mathcal{P}}$ is not an induced subgraph of H.

But every edge uv of G "induces" a copy of J_1 in G' (meaning, if you take both u, v, and a bunch of vertices of A) which together with the copies of $J_2, ..., J_l$ would form a subgraph isomorphic to $J_{\mathcal{P}}$. Furthermore, u and v are the only vertices of that copy of $J_{\mathcal{P}}$ that do not belong to A. Hence, since $J_{\mathcal{P}}$ is an obstruction for $\mathcal{P}$ and since H must contain all the vertices of A, H does not contain u or does not contain v. So $V(H) \cap V(G)$ is a stable set of G which does not intersect B.

So far, we have proven :

- In **1.** that if S is a maximal stable set of G which does not intersect B, then $H := G'[S \cup A]$ is a $\mathcal{P}$-subgraph of G' which contains A and does not intersect B.
- In **2.** that if H is a maximal $\mathcal{P}$-subgraph of G' which contains A and does not intersect B, then $S := V(H) \cap V(G)$ is a stable set of G not intersecting B.

It remains to show that what we obtain in both cases is also maximal.

Let S be a maximal stable set of G which does not intersect B, let us prove that $H := G'[S \cup A]$ is a *maximal* $\mathcal{P}$-subgraph of G' which contains A and does not intersect B. Assume H is not maximal and let $H' \supsetneq H$ be a maximal $\mathcal{P}$-subgraph of G' which contains A and does not intersect B. By **2.**, we know that $S' := V(H') \cap V(G)$ is a stable set of G avoiding B. Then, observe that $S = V(H) \cap V(G) \subseteq V(H') \cap V(G) = S'$. Moreover, there exists $v \in V(H') \setminus V(H)$, consequently $v \notin A$ so $v \in V(G)$. Hence $v \in S' \setminus S$ so S' is a stable set of G that avoids B and strictly contains S, contradicting the maximality of S.

Now, let us prove that if H is a maximal $\mathcal{P}$-subgraph of G' which contains A and does not intersect B, then $S := V(H) \cap V(G)$ is a *maximal* stable set of G which does not intersect B. Let $S' \supseteq S$ be a maximal stable set of G avoiding B which contains S. We will prove that $S' = S$. By **1.** $H' := G'[S' \cup A]$ is a $\mathcal{P}$-subgraph of G' which contains A and does not intersect B. But since $H = G'[S \cup A]$, in particular H is

contained in H', which would contradict the maximality of H unless $S \cup A = S' \cup A$. Since $A \cap S' = \varnothing$, this implies $S' = S$. □

Hence, without any assumptions on the class $\mathcal{P}$, it is not possible to enumerate all maximal induced $\mathcal{P}$-subgraphs of a graph G in polynomial delay using the simple algorithm derived from the extension problem. This, however, does not mean that Flashlight Search cannot be efficient at all, since the sets A and B could in fact be assumed to have certain specific properties depending on $\mathcal{P}$, and which could make the extension problem easy to solve in special cases (see for instance [25, 60]).

4.2 Maximal edge-subgraphs

In the previous section, the extension problem for maximal induced subgraphs was proved to be NP-hard. In the present one, we define the extension problem for maximal edge-subgraphs. This problem is further investigated to try and determine in which cases the extension problem for maximal edge-subgraphs is polynomial, and in which cases it is NP-hard.

Extension problem. For a hereditary graph property $\mathcal{P}$, the *extension problem for maximal edge-subgraphs* is stated as follows. Given a graph $G = (V, E)$ and two disjoint subsets A and B of E, does there exist a minimal $\mathcal{P}$-deletion of G containing all edges of A and no edge of B?

As for the vertex version, it does not suffice in general to look for maximal $\mathcal{P}$-edge-subgraphs of $G - B$, because the $\mathcal{P}$-subgraphs obtained this way may not be maximal $\mathcal{P}$-subgraphs of G.

Recall that a graph property $\mathcal{P}$ is said to be monotone if it is closed under edge removal. In other words, $\mathcal{P}$ is monotone if for any graph G satisfying $\mathcal{P}$ and any edge e of G, $G - e$ also satisfies $\mathcal{P}$.

Such a general proof as in the case of maximal induced subgraphs is hard to obtain when considering maximal edge-subgraphs, even when the property $\mathcal{P}$ is assumed to be monotone. In the sequel, we explore the "edge version" of the extension problem to understand in which cases it can be solved by a polynomial algorithm. Three graph classes are considered: forests, graphs without short induced cycles, and graphs without long induced paths.

4.2.1 Forests

Forests are graphs without induced cycles. They consist in disjoint unions of trees. If $\mathcal{P}$ denotes the class of forests, which is a monotone property, the extension problem asks if there exists a maximal spanning forest of G containing A and avoiding B.

Assume first that G is connected. In this particular case, if removing B first disconnects G, then the answer is no: all spanning trees of G must contain at least one element from B. Else, B can be removed first. The set A, if it induces a forest, can then be extended in $G - B$ to a spanning tree of G. In the case where G is not connected, a (maximal) spanning forest of G is the union of spanning trees of all connected components of G. Therefore, the extension problem for forests is polynomial.

This simple observation has already been made in the article of Read & Tarjan [72] introducing the Flashlight method. They use it to design a very efficient algorithm, running in polynomial delay and polynomial space, for the enumeration of all spanning trees of a connected graph.

Nevertheless, the extension problem for maximal edge-subgraphs is not polynomial for all classes of graphs. In the following two sections, we exhibit two graph properties for which this problem is NP-hard.

4.2.2 P_k-free graphs

For an integer $k \geqslant 3$, let us denote by P_k the path on k vertices. We will now have a look on the class of all graphs that do not have long paths as induced subgraphs: the P_k-free graphs. In particular, when $k = 4$, the class of P_k-free graphs is exactly the class of cographs. Note that the property of being P_k-free is not monotone: a cycle of length k is P_k-free but removing one edge from it yields a P_k.

In 2018, Casel *et al.* proved, among other results, that the extension problem with $A = \varnothing$ is NP-complete in the case of maximal matchings [20]. Since maximal P_3-free graphs correspond exactly to maximal matchings in triangle-free graphs, this result implies that the extension problem for maximal P_3-free subgraphs is NP-complete, even in triangle-free graphs and with $A = \varnothing$. We extend this result in the present section, proving the following theorem.

Theorem 4.2. *The extension problem for maximal P_k-free edge-subgraphs is NP-hard for all $k \geqslant 3$. Moreover, it is NP-complete when k equals 3 or 4.*

Before proving Theorem 4.2, let us see how maximal P_4-free edge-subgraphs can be recognised in polynomial time.

Consider a graph $G = (V, E)$ and an edge-subgraph S of G, identified with its edge set (therefore $S \subseteq E$), since the class of P_4-free graphs is stable under adding isolated vertices. We design a procedure, running in polynomial time, to determine if S induces a maximal P_4-free edge-subgraph of G. The procedure relies on what is called the *sandwich problem*.

The *sandwich problem* [50] is a decision problem defined as follows. We are given a graph property $\mathcal{P}$, and two graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ on the same

vertex set such that $E_1 \subseteq E_2$, which both typically do not satisfy property $\mathcal{P}$. The question is to determine whether there exists a graph $G = (V, E)$ satisfying $\mathcal{P}$ such that $E_1 \subseteq E \subseteq E_2$.

A polynomial algorithm answering this question for cographs – that is, P_4-free graphs – was proposed by Golumbic, Kaplan & Shamir [50]. Such an algorithm is called a *sandwich algorithm* in the sequel. The idea is now to use the sandwich algorithm for cographs to determine if a given edge set produces a maximal P_4-free edge-subgraph.

Recognising a maximal P_4-free edge-subgraph in polynomial time. Consider the graph G and its subgraph S. The "cograph property" can be tested in linear time, so we may assume that S induces a sub-cograph of G.

At the beginning, consider $G_1 = S$ and $G_2 = G$. Since S is a cograph, the sandwich algorithm called on (G_1, G_2) returns TRUE. Then, it suffices to try adding edges to G_1, one at a time, and call the sandwich algorithm: if it returns TRUE on $(G_1 + e, G_2)$ for at least one edge $e \in E \setminus S$, it means that there exists a sub-cograph of G strictly containing S, and S is not maximal. Else, the sandwich algorithm returns FALSE on $(G_1 + e, G_2)$ for each $e \in E \setminus S$, and S is a maximal P_4-free edge-subgraph of G.

The sandwich algorithm for cographs being polynomial, this whole procedure also runs in polynomial time.

Equipped with this polynomial recognition procedure, we are now ready to prove Theorem 4.2.

Proof of Theorem 4.2. For $k = 3$, the result follows from the proof of Casel *et al.* [20] and the previous discussion. Now, let us prove it for $k = 4$.

Let $G = (V, E)$ be a graph, and let $S \subseteq E$ be a set of edges of G. By the previous discussion, knowing if S is a maximal P_4-free edge-subgraph of G is polynomial. Therefore, the problem is in NP. We will now prove that the extension problem for maximal P_4-free edge-subgraphs is NP-hard.

The reduction is from 3-SAT. Begin with a 3-SAT formula, with clauses $C_1, \ldots, C_m$, and variables $X_1, \ldots, X_n$. Consider the graph G built from the formula as follows.

For each clause C_j, build a vertex c_j, and for each variable X_i, build two vertices x_i and $\overline{x_i}$ corresponding to the associated literals, linked by an edge $x_i\overline{x_i}$. Now, add an edge between x_i (or $\overline{x_i}$) and c_j if the corresponding literal appears in clause C_j. At this step, each vertex c_j is of degree 3. Finally, for all j, add a pending vertex c'_j to c_j. Call the resulting graph G. An illustration of such a construction is given in Figure 4.4.

We define $A = \{x_i\overline{x_i} \mid 1 \leqslant i \leqslant n\}$ and $B = \{c_jc'_j \mid 1 \leqslant j \leqslant m\}$. These will be the edge sets A and B considered for the extension problem.

Suppose that there exists a maximal P_4-free subgraph H of G, containing A and avoiding B. Since H is a maximal P_4-free subgraph of G, adding to H an edge of B

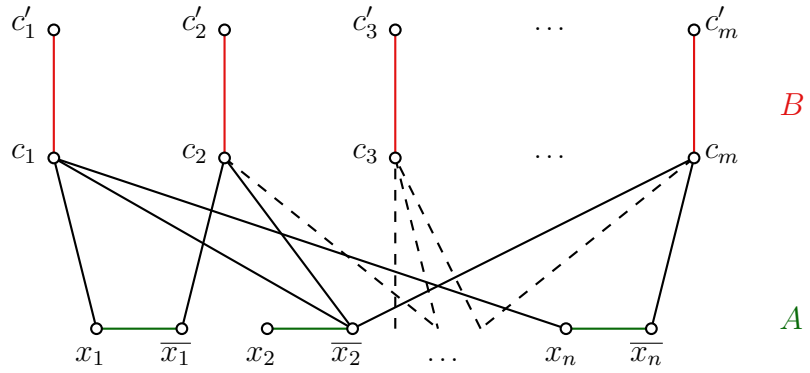

Figure 4.4: The graph G obtained from a 3-SAT formula.

produces an induced P_4. That is to say, each vertex c_j is the endpoint of an induced P_3. Thus, there exists i such that $c_j x_i \in E(H)$ or $c_j \overline{x_i} \in E(H)$. Without loss of generality, suppose that $c_j x_i \in E(H)$ (so $c_j \overline{x_i} \notin E(H)$). In this case, there is no k such that $c_k \overline{x_i} \in E(H)$, otherwise $c_j x_i \overline{x_i} c_k$ is an induced P_4 in H.

For all $1 \leqslant i \leqslant n$, at most one of x_i, $\overline{x_i}$ has neighbours among $\{c_1, \dots, c_m\}$. Assigning the value 1 to the corresponding literal satisfies the formula.

Conversely, if the formula is satisfiable, there exists a maximal P_4-free subgraph H of G containing A and avoiding B. It suffices to keep only edges incident to vertices corresponding to literals assigned the value 1. To avoid creating induced P_4, keep only one edge incident to each vertex c_j. This way, each vertex c_j is the endpoint of an induced P_3, and no edge of B can be added without creating a P_4.

Consequently, the edge extension problem for P_4-free graphs is NP-complete, even in triangle-free graphs because the graph G we built is triangle-free.

It is easy to adapt the previous reduction to show the NP-hardness of the problem for P_k-free graphs, with $k > 4$. It suffices to subdivide each edge $x_i \overline{x_i}$ into a path of length $k - 3$ whose edges are all put in the set A. □

4.2.3 Graphs without cycles of length at most k

Another class of graphs that is worth studying is the class of graphs without cycles of length at most k, for a given $k \geqslant 3$, also known as graphs of girth at least $k + 1$, where the *girth* stands for the length of the shortest cycle. This is a monotone property. For this class of graph, it has been shown by Yannakakis in 1981 that the *edge-deletion problem* is NP-complete [87]. In this case, the edge-deletion problem consists in finding a set of edges of minimum cardinality whose removal results in a graph without cycles of length at most k.

In the same fashion, we prove that the edge extension problem for graphs without cycles of length at most k, that is to say, determining if there exists a maximal edge

subgraph without cycles of length at most k, containing A and avoiding B, is also NP-complete.

Theorem 4.3. *The extension problem for maximal edge-subgraphs without cycles of length at most k is NP-complete for all $k \geqslant 3$.*

Proof. First of all, checking if a set of edges induces a subgraph without cycles of length at most k can be done in time $\mathcal{O}(n^k)$, where n is the number of vertices of the input graph: it suffices to test all possible vertex subsets of size at most k. Moreover, as the property is monotone, maximality can also be checked in polynomial time.

The reduction is from the minimal Vertex Cover extension in triangle-free graphs, whose NP-completeness has been proven by Boros *et al.* [7, Proposition 2]. The minimal Vertex Cover extension problem asks, given a graph G and a subset S of its vertex set, whether there exists an *inclusion-wise minimal* Vertex Cover of G containing S, that is, a subset $X \supseteq S$ of vertices such that each edge has at least one endpoint in X. The original NP-completeness statement [7] does not mention the triangle-free case, but the reduction used in the proof produces only triangle-free input graphs, which proves the NP-completeness we need.

First, we shall prove the result for $k = 3$. Let $G = (V, E)$ be a triangle-free graph, and let $S \subseteq V$. We build another graph $G + c$ by adding a universal vertex c to G. Since G is triangle-free, all triangles in $G + c$ must include c. Moreover, for two vertices $u,\ v \in V$, the three vertices u, v, and c form a triangle in $G + c$ if and only if $uv \in E$. Thus, there is a one-to-one correspondence between the edges of G and the triangles of $G + c$. Finally, we define A and B two subsets of edges of $G + c$ as follows: $A := E(G)$ and $B := \{cv \mid v \in S\}$. An illustration of this transformation is presented in Figure 4.5.

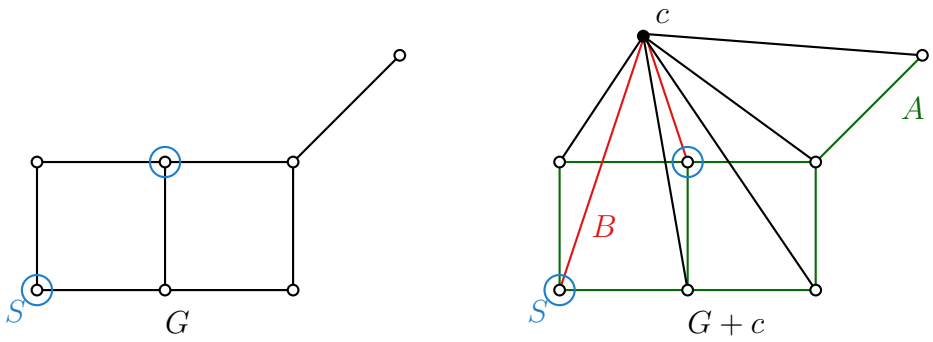

Figure 4.5: Transformation of an instance of Vertex Cover extension into an instance of the triangle-free edge extension problem. Circled vertices are vertices in S, A is the set of edges of G (green), and B is the set of edges between c and vertices of S (red).

We will show that finding an edge extension without cycles of length (at most) 3 is equivalent to finding a minimal Vertex Cover of G containing S, or finding a maximal triangle-free subgraph of $G + c$ containing A and avoiding B.

Let H be a maximal triangle-free subgraph of $G + c$, containing A and avoiding B and let $X = \{v \in V \mid cv \notin E(H)\}$. Let us prove that X is a minimal Vertex Cover of G containing S. Since H is triangle-free, X is a Vertex Cover of G. Indeed, for each edge e of G there exists $v \in e$ such that $cv \notin E(H)$ (hence $v \in X$), otherwise $e \cup \{c\}$ induces a triangle in H.

Moreover, since H does not contain any edge of B, X contains S for otherwise there would be $v \in S$ such that $cv \in E(H)$.

Finally, since H is a maximal triangle-free subgraph of $G + c$, it is easy to see that the Vertex Cover X is minimal in G: if there were $v \in X$ such that $X \setminus \{v\}$ is a Vertex Cover of G, then all neighbours of v (except c) would have to be in X to cover all the edges. So, by definition of X, there is no neighbour u of v such that $cu \in E(H)$, hence we can add the edge cv to H without creating any triangle. This contradicts the maximality of H.

Conversely, if we are given a minimal Vertex Cover of G containing S, then the subgraph of $G+c$ given by $(V \cup \{c\}, A \cup \{cv \mid v \notin S\})$ is triangle-free, and it is maximal in $G + c$ because the Vertex Cover is minimal in G.

This concludes the proof for $k = 3$.

For greater values of k, the proof can easily be adapted by subdividing each edge of G in $G+c$ into a path of length $k-2$ (that is, adding $k-3$ new vertices on each edge), and defining A to contain all edges of those paths. Such a transformation is illustrated in Figure 4.6. Note that for $k > 3$, the smallest cycle in the subgraph induced by A has length at least $3(k-2) > k$, ensuring that the solutions of the extension problem are not trivially inexistent.

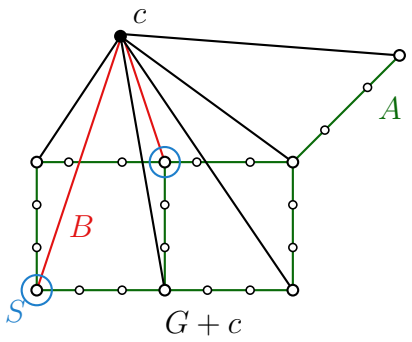

Figure 4.6: Transformation for $k = 5$.

□

As we have seen, in general, solving an extension problem is not easy: in the induced case, the extension problem is NP-hard in general, and in its edge version, there are also some situations in which it is NP-complete. Flashlight Search is efficient to solve many enumeration problems with polynomial delay, but unfortunately the technique is not applicable to *all* of our problems. Therefore, if we want to solve all the maximal subgraphs (induced or not) enumeration problems with a general technique, then Flashlight Search is not adapted. This is why a more sophisticated technique is required.

Chapter 5

Proximity Search: polynomial delay

The results presented in this chapter are obtained by applying and adapting the Proximity Search technique. Parts of them were obtained as a joint work with Aurélie Lagoutte, Vincent Limouzy, Arnaud Mary and Lucas Pastor and appear in the 2020 preprint [10]. Other parts of the results in this chapter (on chordal completions) have been obtained with Vincent Limouzy and Arnaud Mary and published in the proceedings of the 2022 ICALP conference [11].

In this chapter, we study some classes of graphs in which the structure makes it easy to find an elimination ordering. The idea is then to apply Proximity Search: we will use this special, property-dependent ordering as the canonical ordering to define the proximity between two solutions.

The first two classes that will be considered here are *cographs* and *threshold graphs*. Both are well-studied classes of graphs for which it was not known if there exists polynomial delay algorithms enumerating maximal (induced or not) subgraphs of an arbitrary input graph.

Then, we focus on the class of *chordal graphs*. This graph class is very important since it is closely related to graph parameters such as *treewidth* and arises in many applications. A polynomial delay algorithm running in exponential space was already known for maximal induced chordal subgraphs and for minimal chordal deletions [23], but the question of the existence of a polynomial delay algorithm enumerating minimal chordal completions of a graph remained open.

5.1 Maximal induced sub-cographs

As mentioned earlier, the class of cographs is hereditary, and cographs can be recognised in linear time. This implies that finding one maximal induced sub-cograph of an input graph is easy, using a greedy algorithm.

To the best of our knowledge, no polynomial delay algorithm was known for the enumeration of maximal induced sub-cographs of a graph. The idea is to try using Proximity Search on the class of cographs and see if it helps to enumerate them efficiently. Plus, since cographs benefit from a twin construction ordering on their vertex set, as stated in Section 1.1.2, we have a natural candidate for defining a canonical ordering which could be useful in the Proximity Search framework.

If it were possible to solve the input restricted problem (see Section 2.2.4) for maximal induced sub-cographs in polynomial time, then it would yield a polynomial delay algorithm for their enumeration. However, we will show that in our particular case, this problem cannot be solved in polynomial time.

Observe a graph H consisting of k disjoint edges $x_1y_1, \ldots, x_ky_k$. This graph is a cograph. In particular, it does not contain P_4 as an induced subgraph (see Section 1.1.2). Now, add a vertex v adjacent to all vertices x_i for $1 \leqslant i \leqslant k$, as shown in Figure 5.1. The resulting graph $H + v$ is not a cograph any more, since it contains several P_4. Removing one vertex among $\{x_i, y_i\}$ for all $1 \leqslant i \leqslant k$ produces a maximal induced sub-cograph of $H + v$. This means in particular that $H + v$ has at least 2^k maximal induced sub-cographs, which implies that the restricted problem has an exponential number of solutions. Hence, the input restricted problem approach cannot be used for the enumeration of maximal induced sub-cographs with polynomial delay. We will have to use another technique if we want to achieve this time complexity bound.

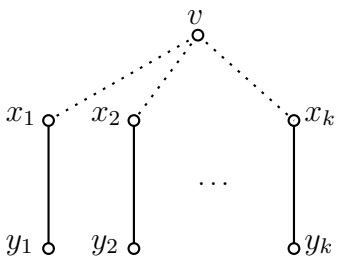

Figure 5.1: The graph H consisting of k disjoint edges $x_1y_1, \ldots, x_ky_k$ is a cograph, and $H + v$ has an exponential number of maximal induced sub-cographs.

Nevertheless, adding one vertex to a cograph produces only a polynomial number of induced P_4. In this case, the results presented by Khachiyan *et al.* in the 2000s [61] can be used to obtain an incremental polynomial time algorithm for the enumeration of maximal induced sub-cographs. Since the size of the obstruction is constant, the

algorithm of Eiter & Gottlob [41] can as well be used to achieve incremental polynomial time complexity. We show here that this time complexity can be improved to polynomial delay with the Proximity Search technique (see Section 2.2.5).

As it was shown by Corneil *et al.* [28], a characterisation of cographs is that they can be constructed from an isolated vertex by recursively adding true or false twins to existing vertices. From such a construction, we obtain an ordering $v_1, \ldots, v_n$ on the vertices of a cograph H, such that for all $j \in \{1, \ldots, n\}$, v_j has at least one (true or false) twin in $H[v_1, \ldots, v_j]$, that we called the *twin construction ordering* in Section 1.1.2.

Every maximal induced sub-cograph can then be identified with an ordered set of vertices; this is exactly what will be done in the sequel. This is why, for simplicity of notation, we sometimes refer to a solution S as a set of vertices instead of a graph (without introducing any ambiguity on the set of edges, since we are only considering induced subgraphs in this section).

Let G be a graph of which we want to enumerate all maximal induced sub-cographs. To use Proximity Search on this problem, we need:

- an ordering scheme on the maximal induced sub-cographs of G;
- a neighbouring function, enabling to generate from one maximal induced sub-cograph S some new maximal induced sub-cographs, among which one will provably be closer to a target solution.

Since the twin construction ordering of a cograph is in general not unique, the first thing to do is to make it canonical. This will be done by means of an arbitrary ordering $u_1, \ldots, u_n$ on the vertices of G, as for many problems solved with Proximity Search [26]. According to the arbitrary ordering just defined, we will say that u_i is smaller than u_j if $i < j$. The ordering scheme we use for maximal induced sub-cographs is defined as follows.

Ordering scheme. For any maximal induced sub-cograph S of G, the canonical ordering $\pi(S)$ is the lexicographically smallest twin construction ordering of S, with respect to the arbitrary ordering $u_1, \ldots, u_n$. In particular, for any S, $\pi(S)$ is a sequence $v_1 \ldots v_k$ of vertices of G such that v_1 is the smallest vertex of G belonging to S.

It is worth mentioning that for any maximal induced sub-cograph S of G, the canonical ordering $\pi(S)$ can be computed in polynomial time from the cotree associated with S, even if this is not required when using the classical Proximity Search framework.

An example of a cograph with its canonical ordering is given in Figure 5.2. In this example, the canonical ordering is $1, 4, 3, 2, 5$.

To build the graph of Figure 5.2, we begin with the vertex of smallest id: 1, then 4 is added as a true twin of 1, followed by 3, a false twin of 1 in the subgraph induced by $1, 4, 3$; after that 2 is added as a true twin of 1, and finally 5 is added as a false twin of the vertex 4.

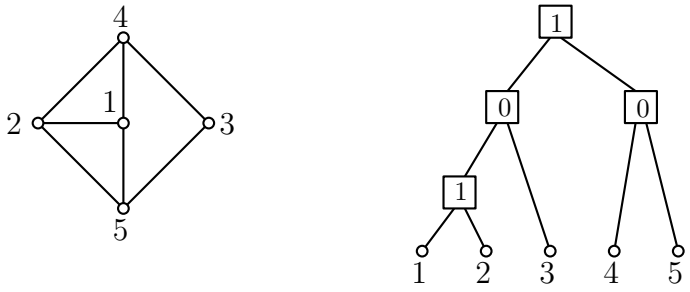

Figure 5.2: An example of a cograph with its cotree; the canonical ordering is $1, 4, 3, 2, 5$.

Equipped with this ordering scheme, we can then define the proximity between two maximal induced sub-cographs of G in the usual manner: if S and S' are two solutions, and the canonical ordering of S' is $v_1 \dots v_{|S'|}$, then $S \widetilde{\cap} S'$ is the longest prefix $v_1 \dots v_i$ of $\pi(S')$ such that $\{v_1, \dots, v_i\} \subseteq S$. In this case, $|S \widetilde{\cap} S'| = i$. Recall that this notion of proximity is not symmetric and that the only solution S maximising $S \widetilde{\cap} S'$ is S' itself.

The next step is to define the neighbouring function NEIGHBOURS that will be used to build the solution graph. Remark that, as the class of cographs is hereditary, the greedy COMPLETE function which adds one vertex at a time while the result is a cograph can be used to maximise an induced sub-cograph. This maximisation function runs in polynomial time because the recognition algorithm for cographs does so. Moreover, it is easy to make this COMPLETE function deterministic by following the arbitrary ordering $u_1, \dots, u_n$.

In order to be able to use Proximity Search, the cardinality of NEIGHBOURS(S) should be polynomial in n for any solution S. We will see that the following neighbouring function satisfies this requirement.

Definition 5.1 (Neighbouring function for maximal induced sub-cographs)**.** *Let S be a maximal induced sub-cograph of G. For all $x \notin S$ and all $y \in S$, we define*

$$S'_{xy} := \text{COMPLETE}((S \setminus (N_G(x) \triangle N_S(y))) \cup \{x, y\}).$$

Then let NEIGHBOURS$(S) := \left\{S'_{xy} \mid x \notin S,\ y \in S\right\}$*. Each solution has at most a quadratic number of neighbours.*

The idea behind this definition of NEIGHBOURS for maximal induced sub-cographs is to add a vertex x while making sure that x and y become twins (true or false according to their adjacency in G) in S'_{xy}. This neighbouring function is illustrated in Figure 5.3, where the circled vertices represent the elements of $(N_G(x) \triangle N_S(y))$. Two of them are removed from S so that x and y become twins in $(N_G(x) \triangle N_S(y))) \cup \{x, y\}$. After that, the induced sub-cograph is maximised.

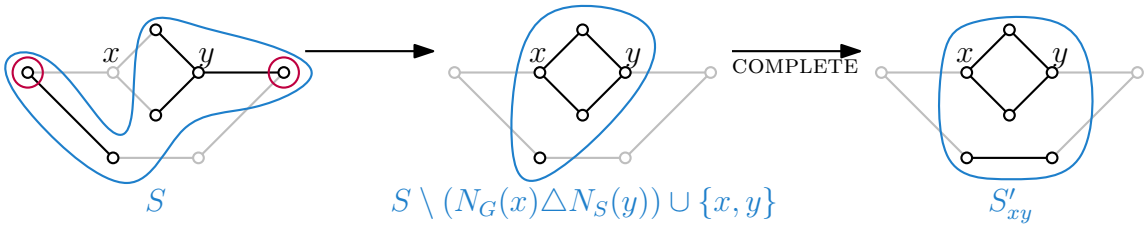

Figure 5.3: Illustration of the neighbouring function for maximal induced sub-cographs. The circled vertices belong to $N_G(x) \triangle N_S(y)$.

Now, we need to ensure that the neighbouring function that has been defined for maximal induced sub-cographs will not generate induced subgraphs that are not solutions to our problem. It is the purpose of the following lemma.

Lemma 5.2. *For every maximal induced sub-cograph S, and for every $x \notin S$, $y \in S$, the graph S'_{xy} is a maximal induced sub-cograph of the input graph G.*

Proof. Let $\hat{S} := (S \setminus (N_G(x) \triangle N_S(y))) \cup \{x, y\}$ and suppose that $\hat{S}$ is not a cograph. Then there is an induced P_4 in $\hat{S}$. Remark that $\hat{S} \setminus \{x\}$ is a cograph, as an induced subgraph of the cograph S. Hence necessarily x participates to this newly-induced P_4, and y does not because it is twin with x. But substituting x for y produces an induced P_4 in $\hat{S} \setminus \{x\}$, which is excluded. Therefore $\hat{S}$ is a cograph.

The function COMPLETE, when given an induced sub-cograph of G, returns a maximal one. Consequently, S'_{xy} is indeed a maximal induced sub-cograph of G. □

Lemma 5.2 guarantees that the neighbouring function is well-defined and that all neighbours of a solution are also solutions. Therefore, we can consider the supergraph of solutions, whose vertices are the maximal induced sub-cographs of G and an arc (S, S') exists if and only if $S' \in$ NEIGHBOURS(S). Now, it remains to show that this supergraph of solutions is strongly connected; in other words, that every solution can be reached from any other by increasing their proximity at each step.

Lemma 5.3. *Let S and S^* be two distinct maximal induced sub-cographs of a given graph G. There exist $x \notin S$ and $y \in S$ such that $|S'_{xy} \widetilde{\cap} S^*| > |S \widetilde{\cap} S^*|$.*

Proof. Denote by $v_1, \ldots, v_{|S^*|}$ the canonical ordering of S^*.

If $|S \widetilde{\cap} S^*| = 0$, then $v_1 \notin S$. Let $y \in S$, then $v_1 \in S'_{v_1 y}$ and it immediately follows $|S'_{v_1 y} \widetilde{\cap} S^*| \geqslant 1 > |S \widetilde{\cap} S^*|$.

Else, let k be the smallest index such that $v_k \notin S$ (in this case, $|S \widetilde{\cap} S^*| = k - 1$). Since $v_1, \ldots, v_k$ is a prefix of the twin construction ordering of S^*, there exists $i < k$ such that v_k and v_i are twins in $S^*[v_1, \ldots, v_k]$. By minimality of k, v_i is a vertex of S.

Consider $S'_{v_k v_i}$. It is a cograph by Lemma 5.2. It remains to show that $S'_{v_k v_i}$ contains $v_1, \ldots, v_k$. For contradiction, assume that there exists $j \in \{1, \ldots, k-1\}$ such that

$v_j \notin S'_{v_k v_i}$. Since $\{v_1, \ldots, v_{k-1}\} \subseteq S \cap S^*$, it implies that $v_j \in N_G(v_k) \triangle N_S(v_i)$. But by definition of the canonical ordering of S^*, v_i and v_k are twins in $G[v_1, \ldots, v_k]$, so no vertex of $\{v_1, \ldots, v_k\}$ can be in $N_G(v_k) \triangle N_G(v_i)$. Therefore, $S'_{v_k v_i}$ contains $v_1, \ldots, v_k$.

Consequently, $|S'_{v_k v_i} \widetilde{\cap} S^*| \geqslant k > |S \widetilde{\cap} S^*|$. □

By Lemma 5.3, the supergraph of solutions is strongly connected. In other words, the problem of enumerating all maximal induced sub-cographs with the twin construction ordering scheme is proximity searchable. Then, Algorithm 2.4 can be used to enumerate all maximal induced sub-cographs in polynomial delay. Hence, we can state the following theorem.

Theorem 5.4. *Maximal induced sub-cographs of a graph G can be enumerated in polynomial delay and exponential space with Proximity Search.*

We also addressed the question of determining if there exists a polynomial delay algorithm computing the minimal cograph completions – or equivalently the minimal cograph deletions, since the class of cographs is self-complementary – of a graph G. As proved in Chapter 4, the corresponding extension problem is NP-complete, so Flashlight Search will not really serve our purposes. As for Proximity Search, finding an appropriate neighbouring function on maximal edge-sub-cographs turned out to be surprisingly difficult. In particular, when trying to adapt the transition function used for maximal induced sub-cographs, we could not make sure that the newly generated subgraphs were cographs. This leads to the following question.

Question 3. *Can minimal cograph deletions be enumerated with polynomial delay?*

5.2 Threshold graphs

In this section, Proximity Search is used to enumerate efficiently the maximal induced threshold subgraphs of a graph G. Unlike for cographs, a good ordering scheme and a good neighbouring function can be derived from the induced case to apply Proximity Search on the minimal threshold deletions. This adaptation of the ordering scheme and neighbouring function to the minimal threshold deletions problem is good news: it implies an efficient algorithm for the enumeration of minimal threshold deletions. Plus, as the class of threshold graphs is self-complementary, it directly implies an efficient algorithm for the enumeration of minimal threshold completions.

5.2.1 Maximal induced threshold subgraphs

By definition, all threshold graphs can be constructed from a single isolated vertex by adding at each step either an isolated vertex or a universal vertex, *i.e.* a vertex

adjacent to all other vertices of the graph [21] (see Section 1.1.2). This construction provides an ordering $v_1, \ldots, v_n$ on the vertices of a threshold graph H, such that for all $1 \leqslant i \leqslant n$, either v_i is isolated in $H[v_1, \ldots, v_i]$, or v_i is universal in $H[v_1, \ldots, v_i]$.

Note that finding a maximal induced threshold subgraph is not hard: a greedy algorithm adding vertices one by one while the resulting graph has the threshold property is well-suited for this purpose. As threshold graphs can be recognised in linear time, the greedy algorithm runs in polynomial time.

We will see how maximal induced threshold subgraphs can be enumerated with the help of Proximity Search. First, an ordering scheme is defined for maximal induced threshold subgraphs, relying on their construction ordering. Then, a neighbouring function allows us to transform a solution into another. The last step is to prove that the maximal induced threshold subgraphs enumeration problem with these ordering scheme and neighbouring function is indeed proximity searchable.

Ordering scheme. As it was the case for cographs, sometimes several threshold construction orderings are possible for a same threshold graph. To define a canonical threshold construction ordering on an induced threshold subgraph S of G, it suffices to fix an arbitrary ordering $u_1, \ldots, u_n$ on the vertices of G and consider the lexicographically smallest threshold construction ordering of S according to $u_1, \ldots, u_n$.

In Figure 5.4 is represented a threshold graph. There are two possible threshold construction orderings of this graph: $1, 4, 2, 5, 3, 6$ and $4, 1, 2, 5, 3, 6$. The lexicographically smallest is $1, 4, 2, 5, 3, 6$. The black vertices are added as universal, and the white ones are added as isolated vertices.

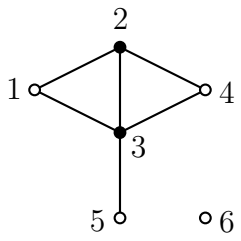

Figure 5.4: The canonical ordering of this threshold graph is $1, 4, 2, 5, 3, 6$.

Now, let us define a neighbouring function able to produce new solutions from the already generated ones.

Definition 5.5 (Neighbouring function for maximal induced threshold subgraphs)**.** *Let G be a graph and let S be a maximal induced threshold subgraph of G. For any $x \in V \setminus S$, we can define two induced subgraphs of G:*

$$S_x^u = \textsc{complete}(\{x\} \cup (S \cap N(x))) \text{ and } S_x^i = \textsc{complete}(\{x\} \cup (S \setminus N(x)))$$

where COMPLETE *is the greedy completion function, adding vertices according to the arbitrary ordering defined a priori, while the threshold property is satisfied.*

Then, let NEIGHBOURS$(S) = \{S_x^u \mid x \in V \setminus S\} \cup \{S_x^i \mid x \in V \setminus S\}$. *Each solution has a linear number of neighbours.*

The neighbouring function for maximal induced threshold subgraphs is illustrated in Figure 5.5. For a vertex x, two induced subgraphs are built: one in which x is universal, and one in which x is isolated. In this example, after the completion step, x is not universal nor isolated any more but the resulting graphs are maximal induced threshold subgraphs of G.

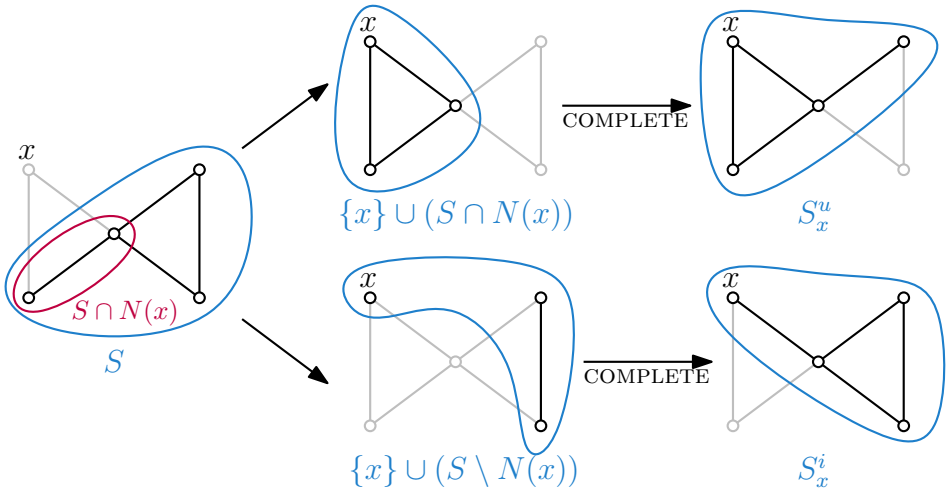

Figure 5.5: Illustration of the neighbouring function for maximal induced threshold subgraphs.

It remains to ensure that the neighbouring function defined here only produces maximal induced threshold subgraphs of G. This is the idea of the following lemma.

Lemma 5.6. *If S is a maximal induced threshold subgraph of a graph G, then for all $x \in V \setminus S$, S_x^u and S_x^i are maximal induced threshold subgraphs of G.*

Proof. Let G be a graph and let S be a maximal induced threshold subgraph of G. It suffices to show that $G[\{x\} \cup (S \cap N_G(x))]$ and $G[\{x\} \cup (S \setminus N_G(x))]$ are threshold graphs. After that, the greedy COMPLETE function will extend them into maximal induced threshold subgraphs of G.

We know that S induces a threshold graph. Moreover, $(S \cap N_G(x))$ and $(S \setminus N_G(x))$ are subsets of S, and since the threshold property is hereditary, they induce threshold subgraphs of S. Now, x is universal to $(S \cap N_G(x))$ so in particular, it is universal in $G[\{x\} \cup (S \cap N_G(x))]$. As the class of threshold graphs is stable under adding a

universal vertex, we immediately deduce that $G[\{x\} \cup (S \cap N_G(x))]$ is a threshold graph. Similarly, x is isolated in $G[\{x\} \cup (S \setminus N_G(x))]$, and since the class of threshold graphs is stable under adding an isolated vertex, then $G[\{x\} \cup (S \setminus N_G(x))]$ is a threshold graph.

Consequently, S^u_x and S^i_x are maximal induced threshold subgraphs of G. □

As before, once we know that all neighbours of a solution are also solutions, it allows us to consider the supergraph of solutions induced by the function NEIGHBOURS. Let us prove that this supergraph of solutions is strongly connected: this way, the problem is proximity searchable and Algorithm 2.4 can be used to enumerate efficiently the maximal induced threshold subgraphs.

Lemma 5.7. *Let S and S^* be two maximal induced threshold subgraphs of a graph G. There exists $x \in V$ such that either $|S^u_x \widetilde{\cap} S^*| > |S \widetilde{\cap} S^*|$ or $|S^i_x \widetilde{\cap} S^*| > |S \widetilde{\cap} S^*|$.*

Proof. Let $v_1, \ldots, v_{|S^*|}$ be the canonical ordering of S^*. Let $k = |S \widetilde{\cap} S^*|$ be the smallest index such that $v_{k+1} \notin S$ (if $v_1 \notin S$, then $k = 0$). By definition of k, we know that $v_{k+1} \notin S$.

Since $v_1, \ldots, v_{k+1}$ is a prefix of S^*, in particular $\{v_1, \ldots, v_{k+1}\} \subseteq S^*$ and the induced subgraph $G[\{v_1, \ldots, v_{k+1}\}]$ is threshold because S^* is. Moreover, by definition of the canonical ordering, we know that v_{k+1} is either isolated or universal in $G[\{v_1, \ldots, v_{k+1}\}]$.

Assume that v_{k+1} is isolated in $G[\{v_1, \ldots, v_{k+1}\}]$. In particular, since we are dealing with induced subgraphs, it implies $\{v_1, \ldots, v_k\} \subseteq S^* \setminus N(v_{k+1})$. Therefore, $S^i_{v_{k+1}}$ contains $\{v_1, \ldots, v_{k+1}\}$, and $|S^i_{v_{k+1}} \widetilde{\cap} S^*| > |S \widetilde{\cap} S^*|$.

Similarly, if v_{k+1} is universal in $G[\{v_1, \ldots, v_{k+1}\}]$, the same reasoning implies the inequality $|S^u_{v_{k+1}} \widetilde{\cap} S^*| > |S \widetilde{\cap} S^*|$. □

From these two lemmas, we deduce that maximal induced threshold subgraphs are proximity searchable, hence the following theorem.

Theorem 5.8. *Maximal induced threshold subgraphs of a graph G can be enumerated in polynomial delay and exponential space with Proximity Search.*

In fact, it has been proved recently that the input restricted problem for maximal induced threshold subgraphs is polynomial [13]. This allows us to use the algorithm proposed by Cohen *et al.* [22] (see Section 2.2.4) to enumerate the maximal induced threshold subgraphs with polynomial delay in polynomial space.

However, being able to enumerate the maximal induced threshold subgraphs with Proximity Search has not become totally pointless. Knowing a suitable ordering scheme for the maximal induced subgraphs problem will help us determine one for the minimal deletions problem, as we will see in the next section.

5.2.2 Minimal threshold deletions

Minimal threshold deletions can also be enumerated with the help of Proximity Search. Since the class of threshold graphs is stable under complement, it is equivalent to enumerate minimal threshold completions as well.

Recall that a minimal threshold deletion of G is a maximal threshold subgraph S of G. Since threshold graphs are closed under adding isolated vertices, it immediately follows that any minimal threshold deletion S satisfies $V(S) = V(G)$. Hence, there will never be any ambiguity on the vertex set, so S will often be used to denote both the subgraph and its set of edges $E(S) \subseteq E(G)$.

Finding one minimal threshold deletion. It has been proved that the class of threshold graphs is *sandwich-monotone* [56]. This means in particular that between two threshold deletions S and T of G such that $T \subseteq S$, there exists a sequence of single-edge additions allowing to go from T to S. But the edgeless graph $T := (V, \varnothing)$ is a threshold subgraph of G: therefore it is a threshold deletion of G such that any minimal threshold deletion S satisfies $T \subseteq S$. Based on this observation, we will show how to find a minimal threshold deletion of G in polynomial time.

Since the edgeless graph is a threshold deletion, it is possible to start from the edgeless graph and try adding one edge at a time while the threshold property is satisfied. We stop when no more edge can be added without breaking the threshold property. Because threshold graphs are sandwich-monotone, we are sure that this procedure will stop only when a maximal threshold edge-subgraph, that is, a minimal threshold deletion of G, is found. Moreover, the threshold property is testable in linear time. Consequently, finding a minimal threshold deletion is possible in polynomial time with this technique.

As in Section 5.2.1, in order to use Proximity Search we need to define an ordering scheme on the solutions (here, on the edges, not on the vertices), and a neighbouring function to go from one solution to some others. The ordering scheme will be adapted from the maximal induced threshold subgraphs case as follows.

Ordering scheme. Let S be a minimal threshold deletion of G. The idea is to order the edges of S in such a way that when a new vertex v_i is added in the threshold construction ordering, all the edges incident to v_i in $S[\{v_1, \ldots, v_i\}]$ are consecutive and follow the threshold construction ordering. More formally, we consider the sequence $((v_j v_i)_{i<j})_{j \leqslant n}$, where $v_1, \ldots, v_n$ is the canonical (*i.e.* lexicographically smallest) threshold construction ordering of S. In fact, we can think of it as adding all edges incident to v_i in $S[\{v_1, \ldots, v_i\}]$ *at the same time*.

The proximity between two minimal threshold deletions S and S^* is then – as usual – the longest prefix of the canonical ordering of S^* included in S.

Since threshold graphs are sandwich-monotone, it is possible to derive a polynomial COMPLETE function for threshold graphs, trying to add one edge at a time while the resulting subgraph is threshold, and stopping when no more edge can be added without breaking the threshold property. Such a completion function returns a maximal induced threshold edge-subgraph, by definition of being sandwich-monotone.

Then, to transform a solution into another, the technique will be to choose a vertex and try to make it universal. We define the following neighbouring function.

Definition 5.9 (Neighbouring function for minimal threshold deletions)**.** *Let G be a graph and let S be a minimal threshold deletion of G. For a vertex x of G, we will build a graph $\widetilde{S}_x$ in which x is universal in the construction ordering. We define $E_G(x) := \{xt \mid t \in N_G(x)\}$, and consider the following sets of edges of G:*

1. *$\widetilde{E}_x(x) := E_G(x)$;*
2. *for each $v \in N_G(x)$, $\widetilde{E}_x(v) := \{vw \mid w \in N_S(v) \cap N_G(x)\} \cup \{vx\}$;*
3. *for every other vertex v, $\widetilde{E}_x(v) := \varnothing$.*

Let $\widetilde{S}_x :=$ COMPLETE $\left(\bigcup_{v \in V} \widetilde{E}_x(v)\right)$, and define NEIGHBOURS$(S) := \{\widetilde{S}_x \mid x \in V\}$*. Each solution has a linear number of neighbours.*

The neighbouring function for minimal threshold deletions is illustrated in Figure 5.6. The vertex x is made universal by adding some edges and removing others. In this example, after the completion step, x is not universal any more but the resulting graph is a minimal threshold deletion of G.

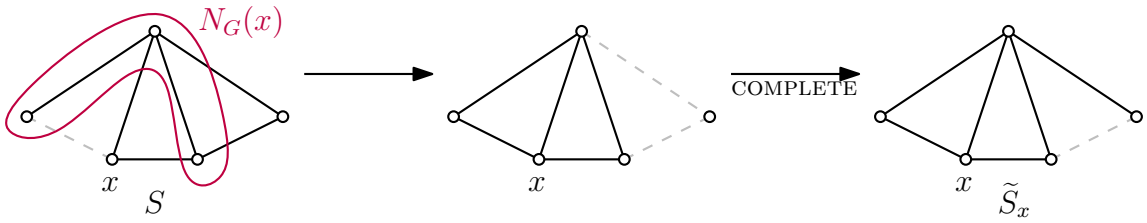

Figure 5.6: Illustration of the neighbouring function for minimal threshold deletions.

As before, we need to ensure that only solutions will be computed by NEIGHBOURS.

Lemma 5.10. *If S is a minimal threshold deletion of a graph G, then for all $x \in V$, $\widetilde{S}_x$ is a minimal threshold deletion of G.*

Proof. Let $x \in V$. It suffices to show that $\widetilde{S} := \bigcup_{v \in V} \widetilde{E}_x(v)$ is a threshold graph.

In $\widetilde{S}$, all non-neighbours of x are isolated vertices. Besides, adjacencies between vertices of $N_G(x)$ have not been modified from S. Moreover, since S is a threshold graph, $S[N_G(x)]$ is also a threshold graph; the addition of x as a universal vertex in $S[N_G(x) \cup \{x\}]$ implies that $S[N_G(x) \cup \{x\}]$ is also a threshold graph. Therefore, as the union of a threshold graph and isolated vertices is a threshold graph, $\widetilde{S}$ is a threshold graph.

It is then possible to apply the polynomial COMPLETE function to build $\widetilde{S}_x$, which is a minimal threshold deletion of G. □

Once we know that all neighbours of a solution are also solutions, we can consider the supergraph of solutions. Let us prove that this supergraph of solutions is strongly connected.

Lemma 5.11. *Let S and S^* be two minimal threshold deletions of a graph G. There exists $x \in V$ such that $|\widetilde{S}_x \,\widetilde{\cap}\, S^*| > |S \,\widetilde{\cap}\, S^*|$.*

Proof. Let $v_1, \ldots, v_n$ be the threshold construction ordering of S^*, and consider the associated edge canonical ordering.

Let $v_k v_i$ be the first edge of S^* which does not appear in S.

By definition, the neighbours of v_k in $\widetilde{S}_{v_k}$ are all the neighbours of v_k in G. Necessarily, we have the inclusion $N_{S^*}(v_k) \subseteq N_{\widetilde{S}_{v_k}}(v_k)$. Hence any edge incident to v_k in S^* is also present in $\widetilde{S}_{v_k}$.

Moreover, if an edge $v_j v_i$ with $j < k$ is present in S^*, then it is also present in S by definition of $v_k v_i$ as the first edge of S^* which does not appear in S. Since $v_k v_i$ is an edge of S^*, it means that v_k is universal in $S^*[v_1, \ldots, v_k]$. Therefore every edge $v_j v_i$ with $j < k$ appearing in S^* is an edge joining two neighbours of v_k in G: it is not removed by the transition operation, so it appears in $\widetilde{S}_{v_k}$.

Hence $\widetilde{S}_{v_k}$ satisfies $|\widetilde{S}_{v_k} \,\widetilde{\cap}\, S^*| > |S \,\widetilde{\cap}\, S^*|$. □

From these two lemmas, we deduce that minimal threshold deletions are proximity searchable, hence the following theorem.

Theorem 5.12. *Minimal threshold deletions (resp. completions) of a graph G can be enumerated in polynomial delay and exponential space with Proximity Search.*

In the case of threshold deletions, it was possible to adapt the canonical ordering and neighbouring function of the maximal induced threshold subgraphs case, in order to apply Proximity Search. Advantage was taken from the "sandwich-monotonicity" of threshold graphs. In fact, being sandwich-monotone (or equivalently, strongly accessible) is a very powerful property, as we will see in the next section.

5.3 Minimal chordal completions

Chordal completions were first introduced in the 1970s as a tool for solving large sparse systems of linear equations [74, 75]. This was inspired by an algorithm known as the *Elimination Game*, proposed by Parter in 1961 and introducing the link between sparse matrix computation and graphs [70]. More precisely, minimal chordal completions are used to compute efficiently what is called the *fill-in* in sparse matrix factorisation. In fact, finding a perfect elimination ordering of a chordal graph corresponds to finding an ordering for the Gaussian elimination of its adjacency matrix [68]. Chordal completions are still used nowadays for this purpose, as they provide indeed a very powerful tool [27]. However, this is far from being the only reason for minimal chordal completions to be interesting.

Being able to compute a tree-decomposition of a graph is useful in many situations. For example, one may want to compute the *treewidth* (minimum size of a bag minus one) since it appears as a parameter in many optimisation problems [73]. But other parameters depend on finding a tree-decomposition, *e.g.* the so-called *tree-independence number* [34]. And the notion of being optimal for a tree-decomposition hugely depends on which parameter has to be optimised (the largest size of a bag, the smallest size of a stable set in a bag, the largest diameter of a bag...) [18, 34]. This is why finding inclusion-wise minimal chordal completions is in some situations even more relevant than finding a minimum-cardinality one (which is NP-complete anyway [86]). Moreover, as the notion of "good" minimal chordal completion is not always easy to model, the enumerative approach makes a lot of sense. This way, one can generate a bunch of minimal chordal completions and choose which one best suits the problem according to their own criteria.

For all these reasons, it is interesting to be able to generate efficiently minimal chordal completions of a graph. For further reference about minimal chordal completions (from another viewpoint than enumeration), we refer the reader to the very complete survey by Heggernes [54].

The question of enumerating all minimal chordal completions of a given graph has been addressed in 2017 by Carmeli, Kenig & Kimelfeld [17], who proposed a highly non-trivial algorithm. Their algorithm runs in incremental polynomial time; its total complexity is quadratic in the number of solutions, and to avoid duplication of solutions, requires exponential space. The algorithm is based on a result by Parra & Scheffler [69], linking the minimal triangulations of a graph to the maximal stable sets of an auxiliary graph. In a later version of their paper, Carmeli *et al.* even proved that, under the Exponential Time Hypothesis, their approach could not be improved to achieve polynomial delay [18]. This intractability result also holds for the exponential space.

In the paper of Carmeli *et al.* [17, 18] and at a Dagstuhl seminar [8], it was left as an open problem whether a polynomial delay or a polynomial space algorithm for the

enumeration of minimal chordal completions could be obtained. Here, we answer this question concerning the time complexity. As for the space usage, it will be treated in Chapter 6.

The other two problems concerning chordal graphs, namely the maximal induced chordal subgraphs and the minimal chordal deletions enumeration, have been addressed by Conte *et al.* in the light of Proximity Search [23]. Both were proved to admit a polynomial delay algorithm, as well as the maximal induced connected chordal subgraphs problem. However, they could not manage to adapt Proximity Search to the minimal chordal completions enumeration.

In the sequel, we exhibit a polynomial delay and exponential space algorithm relying on Proximity Search to list all minimal chordal completions of a graph. First, Section 5.3.1 is devoted to the definition of an ordering scheme that will suit to minimal chordal completions. Then in Section 5.3.2, we introduce the neighbouring function and prove that the problem is proximity searchable. As a consequence, we obtain a polynomial delay algorithm for the enumeration of minimal chordal completions.

5.3.1 Ordering scheme for chordal completions

When looking for an ordering scheme for chordal completions, a property that turns out to be crucial is that of being sandwich-monotone (as it was the case for threshold deletions). It has been proved by Rose, Tarjan & Lueker that the class of chordal graphs is sandwich-monotone [75]. This immediately implies that for any two chordal completions F_1 and F_2 of G, if $F_1 \subseteq F_2$ there exists a sequence of single-edge removals that permits to go from F_2 to F_1. It rephrases as the following lemma.

Lemma 5.13 ([75, Lemma 2])**.** *Chordal graphs are sandwich-monotone. In particular, for any graph G, the chordal completions of G are sandwich-monotone.*

The general idea is then to use the "sandwich-monotonicity" of chordal completions to derive a suitable ordering scheme. Then, we will use this ordering scheme in order to apply Proximity Search.

The Proximity Search framework has been designed to enumerate the inclusion-wise *maximal* sets of a set system. However, the same framework could as well be applied to the enumeration of inclusion-wise *minimal* sets of a set system, by considering the complements of the solutions. It is the approach we adopt for the enumeration of minimal chordal completions.

To this effect, we will not work directly with the edge sets of the completions but rather with the sets of their *non-edges*. For the input graph $G = (V, E)$, let us denote by E^c the set of its non-edges G, that is to say, $E^c = \binom{V}{2} \setminus E$. Here, a chordal completion is identified with its set of fill edges. In other words, we call a chordal completion of

G a set $F \subseteq E^c$, such that the graph $(V, E \cup F)$ is chordal. The set of all chordal completions of G is denoted by $\mathcal{F}$, and the set of all minimal chordal completions of G is denoted by $\mathcal{F}_{\min}$.

Moreover, we consider for any completion F its *complement* $\overline{F}$, defined as $\overline{F} := E^c \backslash F$. The goal of this section is to apply Proximity Search to the set $\overline{\mathcal{F}} := \{\overline{F} \mid F \in \mathcal{F}\}$ in order to enumerate its maximal elements. That is to say, the set we want to enumerate with Proximity Search is $\overline{\mathcal{F}}_{\max} := \max(\overline{\mathcal{F}}) = \{\overline{F} \mid F \in \mathcal{F}_{\min}\}$.

From now on, the elements of E^c (the non-edges of G) are assumed to be arbitrarily ordered. The following definitions are similar to the ones given in the paper by Conte *et al.* [24] but are defined on $\mathcal{F}$ instead of $\overline{\mathcal{F}}$.

Let F be a chordal completion of G and X be any subset of E^c. The set X can be seen as the set from which removing elements is allowed. We define:

- $\textsc{Candidates}(F, X) := \{e \in X \cap F : F \setminus \{e\} \in \mathcal{F}\}$
- $\textsc{Candidates}(F) := \textsc{Candidates}(F, F)$
- $c(F, X) := \min(\textsc{Candidates}(F, X))$
- $c(F) := \min(\textsc{Candidates}(F))$

where the minimum is taken according to the arbitrary ordering on the elements of E^c.

Now, given a chordal completion F and a set $X \subseteq E^c$, we denote by $\textsc{Del}(F, X)$ the chordal completion included in F by iteratively removing $c(F, X)$ from F at each step, until $\textsc{Candidates}(F, X)$ it empty. Finally, we define $\textsc{Del}(F) := \textsc{Del}(F, F)$. Note that, for any F, X, computing $\textsc{Del}(F, X)$ corresponds to the following procedure.

Function 5.1: $\textsc{Del}(F, X)$

```
while Candidates(F, X) ≠ ∅ do
    remove c(F, X) from F;
return F
```

Determining if $\textsc{Candidates}(F, X)$ is empty or computing $c(F, X)$ is not hard: it suffices to try to remove one edge at a time, following the arbitrary order set at the beginning, until one edge is found whose removal leaves the graph chordal, or until all edges have been tested. This can be done in polynomial time because chordality can be tested in linear time [82]. Then, Function 5.1 consists in a polynomial number of calls to this procedure. Consequently, $\textsc{Del}(F, X)$ is computable in polynomial time.

In fact, Function 5.1 has another advantage, stated as the following remark.

Remark 5.14. *By Lemma 5.13, if $F \in \mathcal{F}$ then $\textsc{Del}(F) \in \mathcal{F}_{min}$. That is to say,* Del *can be used to turn a chordal completion into a minimal one in a canonical way.*

Furthermore, note that E^c is a chordal completion of G: it corresponds to the clique completion. This immediately implies, by Remark 5.14, that E^c can be turned into a minimal chordal completion in a deterministic manner. As a result, a minimal chordal completion of G can be found in polynomial time by computing $\textsc{Del}(E^c)$.

Before going further, let us have a look on how the Function Del is used to find a minimal chordal completion of a graph. In the example of Figure 5.7, the input graph G is a C_6, that is, a cycle on six vertices named a, b, c, d, e, f. At first, all the possible fill edges are added. Then, the fill edges are considered in lexicographic order; at each step we try to remove the lexicographically smallest fill edge whose removal does not create a chordless C_4. In this example, at the moment the edge bf is considered, its removal creates a chordless C_4.

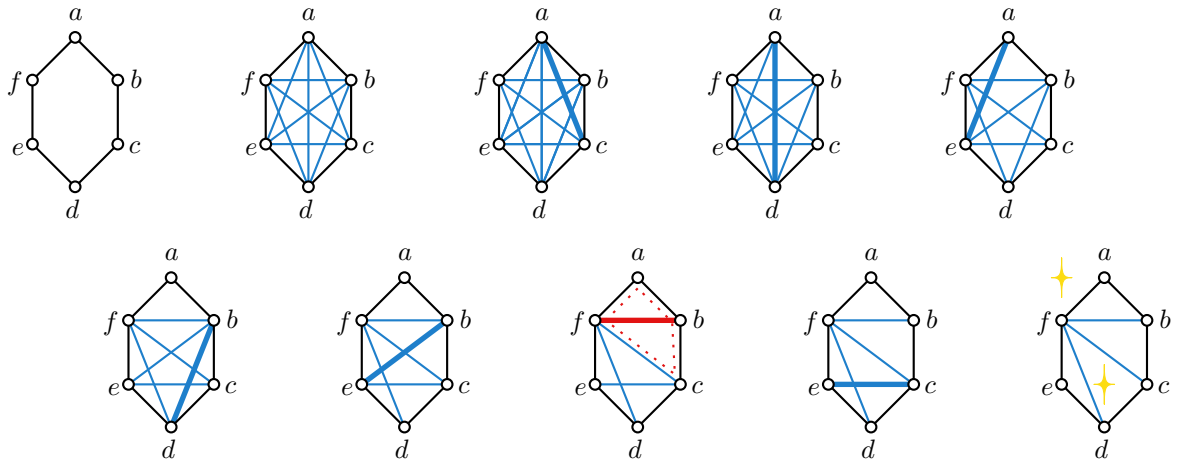

Figure 5.7: How to use the Function Del to compute a minimal chordal completion of a C_6. The chordal completion $\{bf, cf, df\}$ is minimal.

But the fact that E^c is a chordal completion of G has even stronger implications, as stated in the following lemma.

Lemma 5.15. *If F is a minimal chordal completion of G, then $\textsc{Del}(E^c, \overline{F}) = F$.*

Proof. By Remark 5.14, it holds $\textsc{Del}(E^c, \overline{F}) \in \mathcal{F}_{\min}$, and $F \subset \textsc{Del}(E^c, \overline{F})$. As a result, since the only minimal solution containing F is F itself, then $\textsc{Del}(E^c, \overline{F}) = F$. □

In the light of Lemma 5.15, we will now be able to define a suitable ordering on the solutions. Indeed, the procedure followed to compute $\textsc{Del}(E^c, \overline{F})$ provides an ordering on the elements of $\overline{F}$ by considering the order in which the elements of $\overline{F}$ are removed to obtain F. From this ordering, we can define for any chordal completion F the canonical ordering $\textsc{Can}(\overline{F}) := s_1, ..., s_{|\overline{F}|}$ of $\overline{F}$ as follows:

- $s_1 := c(E^c, \overline{F})$;
- for all $1 \leqslant i < |\overline{F}|$, $s_{i+1} := c(E^c \setminus \{s_1, ..., s_i\}, \overline{F})$.

Canonical ordering. For a set $\overline{F}$ and its canonical ordering $\textsc{Can}(\overline{F}) := s_1, ..., s_{|\overline{F}|}$ we define $\overline{F}^i$ as the ith prefix of $\overline{F}$, that is, the set of elements $\{s_1, ..., s_i\} \subseteq \overline{F}$, and we set $F^i := E^c \setminus \overline{F}^i$. By definition of $\textsc{Candidates}(E^c, \overline{F})$, any prefix $\overline{F}^i$ of this ordering belongs to $\overline{\mathcal{F}}$. In other words, F^i denotes the chordal completion of G, not necessarily minimal, obtained by removing the ith prefix of $\textsc{Can}(\overline{F})$ from the clique completion.

This ordering will be used to measure the proximity between two solutions in order to call the Proximity Search algorithm for minimal chordal completions. Note that it can be computed in polynomial time for any solution, the complexity being essentially this of calling Function Del.

We now define the notion of proximity between two solutions that will be used in the sequel. For F_1 and F_2 two minimal completions of G, let $\textsc{Can}(\overline{F_2}) = f_1, ..., f_k$ be the canonical ordering of $\overline{F_2}$. The proximity $\overline{F_1} \widetilde{\cap} \overline{F_2}$ between $\overline{F_1}$ and $\overline{F_2}$ is defined as the largest $i \leqslant k$ such that $\{f_1, ..., f_i\} \subseteq \overline{F_1}$. The notion defined here corresponds to the standard proximity measure that is usually adopted when using the Proximity Search framework [10, 24, 26] and it is not symmetric in general.

Note that the proximity is defined here on the set of elements of E^c (that is to say, non-edges of G) which do *not* belong to the chordal completion. Since we are working on both the completions and their complements, by a slight abuse of language, when we speak about the proximity between two minimal completions, we actually mean the proximity between their complement sets.

5.3.2 Polynomial delay algorithm

We show in this section that the ordering scheme Can defined above is proximity searchable. Since the idea is to consider the complement sets of minimal completions rather than the completions themselves, we will seek to maximise the set of common *non-edges* between two minimal completions in order to increase the proximity. So, we actually show that Can is a proximity searchable ordering scheme of $\overline{\mathcal{F}}$.

The first goal is to define a suitable neighbouring function on the class of chordal graphs. Any solution must have a polynomial number of neighbours, each of them being computable in polynomial time in order to guarantee the polynomial delay when applying Proximity Search.

Given a chordal graph H and an edge $e = \{x, y\} \in E(H)$, the *flip operation* $\textsc{Flip}(H, e)$ consists in removing e from H, and turning the common neighbourhood of x and y into a clique. More formally, if $H' = \textsc{Flip}(H, e)$, then $V(H') = V(H)$ and $E(H') := (E(H) \setminus \{e\}) \cup \{uv \mid u, v \in N_H(x) \cap N_H(y)\}$. The flip operation is illustrated in Figure 5.8.

Since H is chordal, the removal of e can create several chordless C_4 of which e was the only chord. We will see that completing the common neighbourhood of x and y

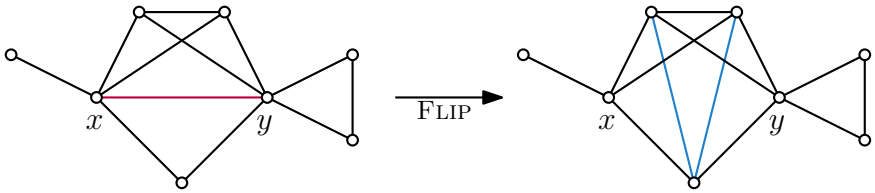

Figure 5.8: The flip operation on $e = \{x, y\}$. The common neighbourhood of x and y is turned into a clique.

into a clique adds the missing chords to all these C_4. In other words, the flip operation preserves the class of chordal graphs, as stated next.

Lemma 5.16. *Let H be a chordal graph, and $e = \{x, y\}$ be an edge of H. Then the graph $H' := \textsc{Flip}(H, e)$ is also chordal.*

Proof. Assume (for contradiction) that H' contains a chordless cycle C of length $\ell > 3$.

Suppose C contains only edges of H. Since H is chordal, it does not contain long induced cycles. Then necessarily C is a C_4 of H of which e was the unique chord, for otherwise there would be an induced cycle of length at least 4 (either C or a cycle made from e and edges of C) in the chordal graph H. By definition, the flip operation adds an edge between the other two vertices of C, meaning that C has a chord in H'. This cannot happen.

Therefore, C contains an edge $e' = \{z, t\}$ added by the flip operation (*i.e.* e' is an edge of H' but not of H). Since C contains the edge e', it does not contain any other fill edge added by the flip operation, otherwise a chord of C would also be added by the flip. Plus, we assumed that e' is added by the flip operation, so necessarily both z and t belong to $N_H(x) \cap N_H(y)$. Therefore, $P := C - \{e'\}$ is a $z - t$ path of H', disjoint from x, y, and e'. Remark that P is an induced path of H since it contains only edges of H, that is, z and t are in the same connected component of $H \setminus \{x, y\}$, as illustrated in Figure 5.9 (left).

Claim 5.17. *Every vertex of P is in $N_H(x) \cap N_H(y)$.*

Proof. Suppose that there exists a vertex s of P which is not a neighbour of x. We will show that there exists in H a chordless cycle containing s and x.

As s is not a neighbour of x, and z is by hypothesis, there exists $q \in N(x)$ a vertex of P that is between z and s, and closest to s on P. Similarly, let $r \in N(x)$ be a vertex of P that is between t and s, and closest to s.

The $q - r$ subpath of P induced by all vertices between q and r is therefore a chordless path of H of which all internal nodes are non-neighbours of x. It follows that adding x to this $q - r$ subpath creates a chordless cycle of length at least 4 in H. The chordless cycle is highlighted on Figure 5.9 (right). This is excluded, so all vertices of P are neighbours of x in H.

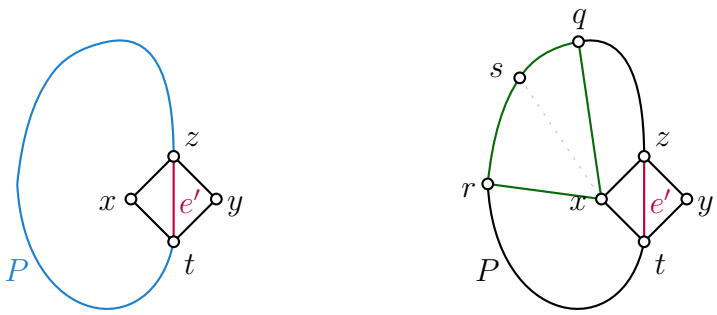

Figure 5.9: There is a $z - t$ path P in H', disjoint from e'. On the right, the vertices q, s, r and x belong to a chordless cycle of H. The edge xs does not belong to the graph.

By symmetry in x and y, we also deduce that all vertices of P are neighbours of y. The claim is proved: every vertex of P is in $N_H(x) \cap N_H(y)$. ▫

By the previous Claim, the flip operation turns the cycle C into a clique. Hence C has length 3, a contradiction. □

When dealing with chordal completions, the notation of the flip operation will be slightly adapted: for $F \in \mathcal{F}$, and $e \in F$ (e is always chosen among the fill edges of the completion), we will omit G in our notation and write $\textsc{Flip}(F, e)$ instead of, for example, $\textsc{Flip}(G \cup F, e)$.

From Lemma 5.16, we are then able to deduce the following equivalent in terms of chordal completions.

Lemma 5.18. *Let F be a chordal completion of a graph G. If $F \in \mathcal{F}_{min}$, and $e \in F$, then $\textsc{Flip}(F, e) \in \mathcal{F}$. That is to say, $\textsc{Flip}(F, e)$ is a chordal completion of G.*

Proof. The only edge that is in F but not in $\textsc{Flip}(F, e)$ is e. Since $e \in F$, it implies that all edges of G are still in $G_{\textsc{Flip}(F,e)}$. As $\textsc{Flip}(F, e)$ is chordal by Lemma 5.16, in particular it is a chordal completion of G. □

We are now ready to explain the neighbouring function used to enumerate $\mathcal{F}_{\min}$. Observe that $\textsc{Flip}(F, e)$ is a chordal completion of G that is not necessarily minimal. It needs to be minimised with Function Del in order to become a solution of our problem.

Neighbouring function. Given $F \in \mathcal{F}_{\min}$ and $e \in F$, we define the *successor* of F according to e as the minimal completion $\textsc{Succ}(F, e) := \textsc{Del}(\textsc{Flip}(F, e))$. We now define the neighbours of a solution F as $\textsc{neighbours}(F) := \{\textsc{Succ}(F, f) | f \in F\}$.

As a corollary of Remark 5.14 and Lemma 5.18, $\textsc{Succ}(F, e) \in \mathcal{F}_{\min}$. Observe that each solution has a polynomial number of neighbours since $|\textsc{neighbours}(F)| \leqslant |F|$.

Now that the neighbouring function is properly defined, it is easy to notice that it can be computed in polynomial time. However, it remains to prove that minimal chordal completions are proximity searchable. To achieve this we need to prove that the supergraph of solutions defined by the relation NEIGHBOURS is strongly connected. This will allow us to use Proximity Search on the set of minimal chordal completions of a graph G.

Lemma 5.19. *Let F_1 and F_2 be two minimal chordal completions of a graph G. Let $f_1, \ldots, f_k = \textsc{Can}(\overline{F_2})$ and let $i := \overline{F_1} \widetilde{\cap} \overline{F_2}$. Then the following statements hold:*

1. $f_{i+1} \notin \overline{F_1}$;
2. $\overline{\textsc{Succ}(F_1, f_{i+1})} \widetilde{\cap} \overline{F_2} > i$.

Proof. 1. By definition of i as the length of the largest prefix of $\textsc{Can}(\overline{F_2})$ included in $\overline{F_1}$, it holds $f_{i+1} \notin \overline{F_1}$.

2. Let $F' := \textsc{Flip}(F_1, f_{i+1})$, we will show that F' increases the proximity with F_2 by proving the inclusion $\{f_1, ..., f_{i+1}\} \subseteq \overline{F'}$. Since $\textsc{Succ}(F_1, f_{i+1}) \subseteq F'$, it implies that $\{f_1, ..., f_{i+1}\} \subseteq \overline{\textsc{Succ}(F_1, f_{i+1})}$, and finally $\overline{\textsc{Succ}(F_1, f_{i+1})} \widetilde{\cap} \overline{F_2} \geqslant i + 1$.

First, notice that by definition of FLIP, $f_{i+1} \notin F'$. Let x and y be the two endpoints of f_{i+1}. Assume for contradiction that an edge f_j, $j \leqslant i$ is added when completing $N_{G_{F_1}}(x) \cap N_{G_{F_1}}(y)$ into a clique. As stated earlier, the notation $\overline{F_2}^{i+1}$ is used to denote the set $\{f_1, ..., f_{i+1}\}$, which is included in $\overline{F_2}$, and F_2^{i+1} represents the associated chordal completion.

Let u and v be the two endpoints of f_j. Then x, u, y, v forms a C_4 of which f_{i+1} was the unique chord in G_{F_1}. By definition of $\textsc{Can}(\overline{F_2})$, the $(i+1)$th prefix F_2^{i+1} is a chordal completion of G, and since neither the chords f_j nor f_{i+1} belong to it, at least one of the edges xu, uy, yv or vx does not belong to F_2^{i+1} since otherwise x, u, y, v would form a chordless C_4 in F_2^{i+1}. Assume without loss of generality that $xu \notin F_2^{i+1}$. Since $\overline{F_2}^{i+1} = \{f_1, ..., f_{i+1}\}$, there exists $\ell \leqslant i$ such that $xu = f_\ell$. But then we have found an edge $f_\ell \notin \overline{F_1}$ with $\ell \leqslant i$, which contradicts the assumption $\overline{F_1} \widetilde{\cap} \overline{F_2} = i$.

Consequently, $\overline{\textsc{Succ}(F_1, f_{i+1})} \widetilde{\cap} \overline{F_2} > i$. □

As a consequence, since the polynomially computable function SUCC is able to increase the proximity between two solutions, the ordering scheme defined by CAN is proximity searchable. More formally, the set of minimal chordal completions of a graph G, with the ordering scheme CAN and the function NEIGHBOURS defined on $\overline{\mathcal{F}}$ by $\textsc{neighbours}(\overline{F}) := \{\overline{X} \mid X \in \textsc{neighbours}(F)\}$ is proximity searchable.

Therefore, Algorithm 2.4 can be used to enumerate $\overline{\mathcal{F}}_{\max}$ or equivalently $\mathcal{F}_{\min}$ with polynomial delay. Yet, applying Proximity Search usually gives polynomial delay with exponential space: to avoid duplication it is necessary to store all the generated solutions in a lookup table. Each newly generated solution is then searched in the

table in polynomial time. The complexity results achieved so far on minimal chordal completions are summarised in the following Theorem.

Theorem 5.20. *There exists a polynomial-delay algorithm for the enumeration of minimal chordal completions of a graph, running in exponential space.*

Remark that, to prove minimal chordal completions are proximity searchable, we needed in the proof of Lemma 5.19 a special property satisfied by the ordering scheme CAN. Indeed, in order for the proof to be correct, the prefixes of a solution must induce *partial solutions*, here non-necessarily minimal chordal completions of G. In the present case, this property is intimately linked with the "sandwich-monotonicity" of chordal completions. Although this requirement seems in a sense quite natural, it is not always satisfied: for example, the ordering scheme we used for minimal threshold deletions does not have this property.

Proximity Search can be used on all the problems we considered in this chapter. This powerful technique allows us to enumerate maximal induced sub-cographs and threshold subgraphs, as well as minimal threshold and chordal completions with a remarkable efficiency. The main drawback of Proximity Search is that in general it uses exponential space: one must store all generated solutions in order to determine if the newly found one has already been generated in the past.

The next goal is to find a general technique able to achieve both polynomial delay *and* polynomial space for many enumeration problems. This is why in the next chapter, we will expose a new technique that allows us to achieve polynomial space for all the problems we solved here with Proximity Search, and many others.

Chapter 6

Proximity Search extensions

In this chapter, we reduce the space usage of some enumeration problems solved with Proximity Search. To this effect, we propose some extensions of the framework. Parts of the results of this chapter appear as joint work with Vincent Limouzy and Arnaud Mary in the proceedings of the 2022 ICALP conference [11].

The Proximity Search framework has proved efficient concerning the enumeration of maximal (induced or not) subgraphs satisfying a property $\mathcal{P}$. However, this technique presents some limitations, in particular on the space usage of the algorithms based on it. This is why the present chapter is dedicated to the formulation of some extensions of the Proximity Search framework.

First, we formalise the idea that was used to derive an ordering scheme for minimal threshold deletions from the one used for maximal induced threshold subgraphs in Chapter 5. In fact, we show that having an ordering on the elements of a solution is not strictly necessary: it is possible to add a bunch of elements at the same time while keeping a good notion of proximity.

Then, in the light of the general framework newly defined, we introduce a new method called *Canonical Path Reconstruction*, aiming to reduce the space usage of Proximity Search on several enumeration problems. This technique consists in the polynomial-time computation of a function Parent with which it is possible to define a spanning arborescence on the supergraph of solutions in the manner of Reverse Search.

In Sections 6.1 and 6.2, we set ourselves in the more general framework of *set systems*, where Proximity Search is meant to be applied. The name *set system* simply denotes a family of subsets – often denoted by $\mathcal{F}$ – of a ground set U.

In fact, induced subgraphs and graph deletions, as well as graph completions (even if this last case is a little bit less obvious) correspond to particular set systems. An

induced subgraph can be seen as a set of vertices of the input graph, and a graph deletion – provided the graph property under consideration is closed under adding isolated vertices, which is always the case for the graph properties studied in this thesis – can be seen as a set of edges. There is an obvious bijection between the set of all (induced or not) $\mathcal{P}$-subgraphs, for a property $\mathcal{P}$, and the family of all sets of vertices (resp. edges) inducing $\mathcal{P}$-subgraphs.

What we enumerate with Proximity Search are the inclusion-wise maximal elements of a set system. Sometimes, the set system is closed under inclusion (as for induced subgraphs satisfying a hereditary property), but in general it is not the case. Think for example about the minimal chordal completions of a graph, for which the corresponding set system is not hereditary.

Therefore, considering the elements of a solution one by one may not be the most accurate approach: sometimes, it is better to add a set of elements and to treat it as a whole, than to split it arbitrarily to fit an element-by-element framework. For this reason, we introduce in the next section a generalisation of the Proximity Search technique.

6.1 General set proximity

In certain cases, we are not able to define an interesting ordering on the elements of a solution, *e.g.* when dealing with maximal edge-subgraphs that satisfy a certain property. Recall when Proximity Search has been applied to enumerate minimal threshold deletions in Chapter 5: the canonical ordering was adapted from the induced subgraphs case to simulate the simultaneous addition of several edges. In this situation, as in some others, we would prefer to build a solution by adding at each step not one element but a bunch of elements all at the same time.

However, this cannot be done directly with Proximity Search since this method precisely relies on the encoding of solutions as *ordered sequences of elements*. To cope with this limitation, we propose here a generalisation of Proximity Search. Instead of considering an ordering on the elements of the ground set, we define an ordering scheme as an ordered sequence of *disjoint subsets* of the ground set.

Let us consider a set system $\mathcal{F}$ on a ground set U. We are interested in enumerating the set $\mathcal{F}_{\max}$ of all inclusion-wise maximal elements of $\mathcal{F}$.

Definition 6.1 (Set-ordering scheme)**.** *A* set-ordering scheme Π *of* $\mathcal{F}$ *is a function that associates to each* $F \in \mathcal{F}$ *an ordered partition* $\Pi(F) = E_1, \dots, E_k$ *of* F *such that for any* $F \in \mathcal{F}$ *and any* $i \leqslant k$*, there exists a family* $\mathcal{T}_i$ *(depending on* $E_1, \dots, E_{i-1}$*) of subsets of* U*, called* admissible sets*, with* $E_i \in \mathcal{T}_i$*.*

In particular, being a partition of F means that $\bigcup_{i=1}^{k} E_i = F$, and for any $i \neq j$, $E_i \cap E_j = \varnothing$. Interestingly, this new notion of ordering scheme could even be considered on set systems that are not *accessible*[1], but satisfy weaker conditions.

With this definition of a set-ordering scheme, an ordering scheme in the usual sense (defined in Section 2.2.5) is a set-ordering scheme where at step i, the sets E_j, $j \leqslant i$, are in fact the singletons $\{v_j\}$, and we have $\mathcal{T}_i = \{\{v\} \mid v \notin \{v_1, \ldots, v_{i-1}\}\}$.

Ordering schemes can possess some nice properties. For example, it often happens that for a given (set-)ordering scheme, all prefixes of a solution induce partial solutions, that is, non-necessarily maximal elements of the family $\mathcal{F}$. Note that this property is automatically satisfied when dealing with hereditary set systems, as a prefix is in particular a subset. This corresponds to the "natural" property that was used in the proof of Lemma 5.18 when showing that the enumeration of minimal chordal completions was indeed proximity searchable. It is called the *prefix property* and is in fact a typical requirement for Proximity Search-based techniques such as *canonical reconstruction* [23].

Definition 6.2 (Prefix property)**.** *A set-ordering scheme* Π *of* $\mathcal{F}$ *has the* prefix property *if for any* $F \in \mathcal{F}$ *with* $\Pi(F) = E_1, \ldots, E_k$*, for any* $i \leqslant k$*,*

$$\bigcup_{j=1}^{i} E_j \in \mathcal{F}.$$

In other words, all prefixes of $\Pi(F)$ *induce partial solutions (non-necessarily maximal elements of* $\mathcal{F}$*).*

When the ordering scheme for threshold deletion was adapted from the induced subgraphs case, the principle was the following. We considered a solution S as a set of edges, and started from the edgeless graph on $V(G)$. Then, we simulated the addition of a new vertex v_k by adding all edges between v_k and the v_i, $i < k$, at the same time (in an arbitrary order). In fact, it corresponds to this new notion of set-ordering scheme.

At step k, the set $\mathcal{T}_k$ of admissible sets is the collection of all neighbourhoods *in* $\{v_1, \ldots, v_{k-1}\}$ of the remaining vertices, that is,

$$\mathcal{T}_k = \{R \subseteq \{vw \mid w \in N_G(v) \cap \{v_1, \ldots, v_{k-1}\}\} \mid v \notin \{v_1, \ldots, v_{k-1}\}\}.$$

Then, $E_k = \{v_k w \mid w \in N_S(v_k) \cap \{v_1, \ldots, v_{k-1}\}\}$. Moreover, the graph on $V(G)$ with edge set $E_1 \cup \ldots \cup E_k$ obtained at step k is a threshold graph, since it corresponds to the graph induced in S (a threshold graph) by $\{v_1, \ldots, v_k\}$, plus some isolated vertices.

[1] A quite weak assumption on a set system: it is *accessible* if for any non-empty member of the family, there exists an element that can be removed while staying in the family, or in more mathematical terms, $\forall X \in \mathcal{F},\ X \neq \varnothing,\ \exists x \in X,\ X \setminus \{x\} \in \mathcal{F}$. All the set systems corresponding to sandwich-monotone graph properties are accessible.

Therefore, adapting the usual ordering scheme to threshold deletions produces in fact a set-ordering scheme having the prefix property.

In fact, one can go further. When deriving the set-ordering scheme on threshold deletions, what was actually used is that the class is hereditary and closed under adding isolated vertices. Therefore, such a set-ordering scheme can be constructed for other graph properties, *e.g.* for cograph deletions. We state the following observation.

Observation 6.3. *For a hereditary graph class $\mathcal{P}$, closed under adding isolated vertices, with a "usual" ordering scheme for induced $\mathcal{P}$-subgraphs, a set-ordering scheme with the prefix property can always be derived for $\mathcal{P}$-deletions.*

We have seen that set-ordering schemes extend the classical notion of ordering scheme. Let us now look at how the other aspects of Proximity Search are adapted in the general set framework. In particular, the new notion of set-ordering scheme implies a new definition of the proximity on solutions seen as ordered partitions rather than sequences.

Proximity. Given two sets F and $F' \in \mathcal{F}_{\max}$ with $\Pi(F') = E_1, \ldots, E_k$, the *proximity* $F \mathbin{\widetilde{\cap}} F'$ between F and F' is the largest *index* $i \leqslant k$ such that $E_j \subseteq F$ for all $j \leqslant i$, or in other terms, $\bigcup_{j=1}^{i} E_j \subseteq F$.

With this definition, the proximity is not a set any more (as it was the case in the original Proximity Search), but a *number*. When the sets E_i are singletons, this new proximity corresponds to the cardinality of the "classical" proximity. In practice, this does not change a lot from the classical framework, since Proximity Search is about increasing the proximity, which means "increasing the number of elements in the proximity" if the proximity is defined as a set. Therefore, using the corresponding number to define the proximity does not lose any useful information compared to the usual definition.

For simplicity of notation, we keep the same symbol $\widetilde{\cap}$ as before to denote the proximity. This is why, unlike in Chapter 5, for any two solutions F and F', the neighbouring function will not increase $|F \mathbin{\widetilde{\cap}} F'|$ but simply $F \mathbin{\widetilde{\cap}} F'$.

In fact, with this newly defined proximity notion, the definition of a proximity searchable ordering scheme remains almost the same. The only real difference lies in the definition of the proximity function used in practice. We recall here what it means for a problem to be proximity searchable, according to the definition given in the original paper [26].

Definition 6.4 (Proximity searchable)**.** *An enumeration problem is called* proximity searchable *if one solution can be computed in polynomial time, and there exists a proximity function $\widetilde{\cap}$ and a polynomial-time-computable function* NEIGHBOURS *such that for any two solutions F, $F^* \in \mathcal{F}_{\max}$, there exists $F' \in$ NEIGHBOURS(F) satisfying $F' \mathbin{\widetilde{\cap}} F^* > F \mathbin{\widetilde{\cap}} F^*$.*

Therefore, our framework corresponds to a more general case of Proximity Search than the one usually considered.

By definition, a polynomial-time computable function NEIGHBOURS, able to increase the proximity to a target solution, induces a path between any two solutions in $\mathcal{F}_{\max}$. This guarantees that the supergraph of solutions is strongly connected – provided that the neighbouring function is well-chosen – and that the solution space can be explored in a DFS manner with Algorithm 2.4. This can be stated as follows, as an analogous of the original Proximity Search theorem [26].

Theorem 6.5 (Set Proximity Search)**.** *For a given enumeration problem, if there exists a set-ordering scheme and a polynomial-time-computable function* NEIGHBOURS *able to increase the proximity to a target solution, then the enumeration problem admits a polynomial delay enumeration algorithm.*

The novelty of our approach is to measure the proximity according to an ordering scheme on *element sets*, and not just single elements. In particular, with this definition, the set of all known proximity searchable problems is enlarged. In this sense, the general set proximity framework extends the usual Proximity Search.

Now that the framework is set up, it is time to see which improvements can be made in order to achieve better space complexity.

6.2 Canonical path reconstruction

In this section, we consider a set system $\mathcal{F}$ on the ground set U, and we assume that the problem of enumerating the set $\mathcal{F}_{\max}$ of all maximal elements of $\mathcal{F}$ is proximity searchable. In particular, this means that there exists Π a set-ordering scheme on $\mathcal{F}$, a function NEIGHBOURS able to increase the proximity to a target solution, and $F_0 \in \mathcal{F}_{\max}$ a "reference solution" that can be found in polynomial time. The goal is to design a new Proximity Search-based technique achieving polynomial space.

Some work has already been done about the space complexity of enumeration with Proximity Search. In the original Proximity Search paper [23], the authors present a technique that allows them to enumerate the maximal elements of a set system, provided this set system is *commutable*, that is, satisfies the following definition.

Definition 6.6 (Commutable set system)**.** *A set system $\mathcal{F}$ is called* commutable *if it is strongly accessible and satisfies the* commutable property*. Formally, a commutable set system satisfies the following two conditions.*

***strongly accessible*:** $\forall X, Y \in \mathcal{F},\ X \subseteq Y,\ \exists y \in Y \setminus X,\ X \cup \{y\} \in \mathcal{F}$;

***commutable property*:** $\forall X, Y \in \mathcal{F},\ X \subseteq Y,\ \forall a, b \in Y \setminus X,$
$(X \cup \{a\} \in \mathcal{F}) \wedge (X \cup \{b\} \in \mathcal{F}) \implies X \cup \{a, b\} \in \mathcal{F}.$

Intuitively, being commutable means that if two elements can be added separately while staying in the family, then both can be added at the same time. This requirement, however, is quite strong: many set systems are not commutable. For example, although threshold graphs are strongly accessible (see Section 5.2), the commutable property is not satisfied by threshold graphs, as illustrated in Figure 6.1.

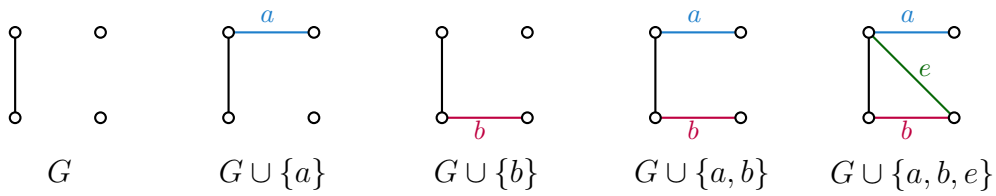

Figure 6.1: The set of threshold graphs is not commutable. In this illustration, G is threshold, $G \cup \{a\}$ and $G \cup \{b\}$ are also threshold, but $G \cup \{a, b\}$ is not (while $G \cup \{a, b, e\}$ is).

By definition, commutable set systems are in particular strongly accessible (but we have just seen that the converse is not true). Moreover, hereditary set systems are commutable. A summary of the inclusions between the different properties of set systems is presented in Figure 6.2.

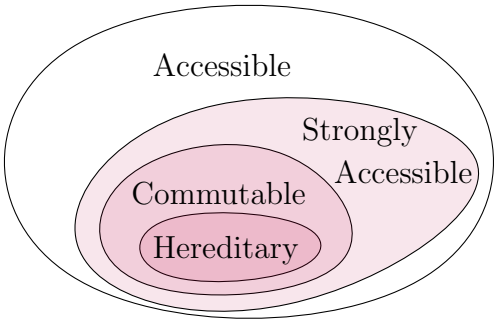

Figure 6.2: Summary of the inclusions between the different properties of set systems.

The idea now is to translate the Reverse Search arborescence into our general set proximity framework, and determine under which conditions this arborescence actually exists.

Since the problem under consideration is assumed to be proximity searchable, for any $F \in \mathcal{F}_{\max}$, there exists a path from F_0 to F in the supergraph of solutions. It is then possible to consider a function NEXT which for any two solutions F and F^* in $\mathcal{F}_{\max}$, chooses deterministically a solution $F' \in \text{NEIGHBOURS}(F)$ such that $F' \mathbin{\widetilde{\cap}} F^* > F \mathbin{\widetilde{\cap}} F^*$.

The deterministic choice of the solution increasing the proximity with F^* can be seen as the following procedure. First, define a preference order, denoted by $<_{\text{NEXT}}$, on

the set of neighbours of the current solution. Then, select the minimum one according to $<_{\text{NEXT}}$. The typical – but not mandatory – choice for Function NEXT will be to select the lexicographically smallest (according to an arbitrary ordering on the ground set) such solution F', that is, $<_{\text{NEXT}} = <_{lex}$.

Remark that for any two solutions F and $F^* \in \mathcal{F}_{\max}$, $\text{NEXT}(F, F^*)$ can be computed in polynomial time. Indeed, to compute $\text{NEXT}(F, F^*)$, it suffices to compute $\text{NEIGHBOURS}(F)$, and for each solution F' in $\text{NEIGHBOURS}(F)$, compute the proximity between F' and F^* and check whether F' is smaller than the current best candidate for the ordering $<_{\text{NEXT}}$ (notice that we do not need to compute $\Pi(F')$).

Let f denote the size of a largest solution, and $\mathcal{N}$ be the complexity of the (polynomial) Function NEIGHBOURS. Then the two required computations can be done in $O(f)$, and since $|\text{NEIGHBOURS}(F)| < \mathcal{N}$, the Function NEXT can be computed in time $O(f\mathcal{N})$.

Canonical path. Equipped with this function NEXT, we can now define for any solution $F \in \mathcal{F}_{\max}$ the *canonical path* of F as the sequence $F_0, \ldots, F_k = F$ of solutions such that for each $1 \leqslant i \leqslant k$, $F_i = \text{NEXT}(F_{i-1}, F)$.

In particular, with the definition we gave for Function NEXT, the solutions F_i on the canonical path of F get closer and closer to F as i increases. This can be restated as the following observation.

Observation 6.7. *Let $F \in \mathcal{F}_{\max}$ and let $F_0, \ldots, F_k$ be its canonical path. Then for any $i < j \leqslant k$, $F_j \,\widetilde{\cap}\, F > F_i \,\widetilde{\cap}\, F$.*

If the solutions on the canonical path of F are closer and closer to F, then in particular, there cannot be too many solutions on this canonical path. Their number is bounded by the number of elements in F, plus one if the root solution F_0 had no common element with F. This can be formulated as the immediate following Corollary, ensuring that canonical paths are of polynomial size.

Corollary 6.8. *If $F \in \mathcal{F}_{\max}$, then its canonical path is of length at most $|F| + 1$.*

Note that the canonical path is not necessarily the path that is followed when exploring the supergraph of solutions with Algorithm 2.4. A solution could be found in a totally different way. In fact, except for F_{k-1}, the other ancestors of F in the exploration arborescence may not be part of its canonical path. For instance, there is no reason for F_{k-1} to be generated from F_{k-2} in the enumeration process. However, the canonical path is uniquely determined for each solution $F \neq F_0$. This will help us define a suitable parent-child relation over $\mathcal{F}_{\max}$ which does not require the system to be commutable, in order to obtain a polynomial space algorithm.

Parent relation. In the context of canonical path reconstruction, the *parent* of a solution $F \neq F_0$, denoted $\textsc{Parent}(F)$, is defined as the last solution before F in its canonical path. In other words, if the canonical path of F is the sequence $F_0, \ldots, F_k = F$, then $\textsc{Parent}(F) = F_{k-1}$. With this Parent function, we can define the set $\textsc{Children}(F) = \{F' \in \textsc{neighbours}(F) \mid \textsc{Parent}(F') = F\}$ of *children* of a solution F.

Note that, in order to achieve polynomial delay, it is necessary to be able to compute the parent of any solution (except F_0 that does not have a parent) in polynomial time. With our definition, this is in fact the case: for any solution F, its parent $\textsc{Parent}(F)$ can be computed in polynomial time, provided that neighbours and the ordering Π can be computed in polynomial time. The complexity of computing the parent of a solution is detailed in Lemma 6.9.

Lemma 6.9. *The function* Parent *can be computed in time $O(p + f^2\mathcal{N})$ where f is a maximum size of a solution in $\mathcal{F}_{\max}$, $\mathcal{N}$ is the complexity of function* neighbours *and p is the time needed to compute Π.*

Proof. To compute $\textsc{Parent}(F)$, we first need to compute $\Pi(F)$ and then to compute the canonical path of F. By Corollary 6.8, the path is of length at most $|F|$, so we only need to call $|F|$ times the function Next. As mentioned previously, this last function can be computed in time $\mathcal{O}(f\mathcal{N})$, whence the result. □

For any solution F, since the computation of $\textsc{Children}(F)$ is done by computing first the set $\textsc{neighbours}(F)$ and then filtering it according to the function Parent, we obtain the following complexity for the function Children.

Corollary 6.10. *The function* Children *can be computed in time $O(p + f^2\mathcal{N}^2)$ where f is a maximum size of a solution in $\mathcal{F}_{\max}$, $\mathcal{N}$ is the complexity of function* neighbours *and p is the time needed to compute Π.*

The parent relation induces a spanning subgraph of the supergraph of solutions. However, if we are not lucky enough, it can happen that this relation induces cycles. Indeed the canonical path of a solution is not necessarily the same as the canonical path of its parent. In particular, this implies, if we decide to explore the supergraph of solutions by following only the transitions authorised by the function Parent, some solutions could never be attained. Such a situation is illustrated in Figure 6.3, where the three solutions P, Q, and R are the parent of each other. They cannot be reached from F_0 by following the parent relation. Therefore, if no further assumption is made on the problem, nothing can be said about the space complexity of Proximity Search with canonical path reconstruction.

Logically, our goal is now to find out under which assumptions the function Children induces an arborescence on the solution space. Then, using a technique similar to Reverse Search, it will be possible to follow the arborescence to enumerate $\mathcal{F}_{\max}$ with polynomial delay and polynomial space.

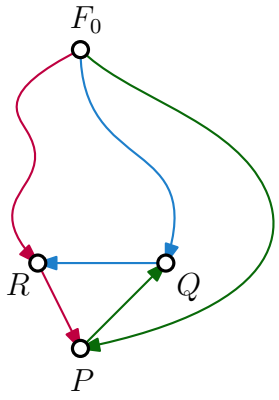

Figure 6.3: The parent relation induces a cycle in the supergraph of solutions. Three solutions have disjoint canonical paths but they are the parent of each other. In this situation, the solutions P, Q, and R cannot be obtained by following the parent relation starting from F_0.

Prefixes. Given $F \in \mathcal{F}$, $F = E_1, \ldots, E_k$, we define F^i the ith prefix of $\Pi(F)$ as $F^i = E_1, \ldots, E_i$ for all $i \leqslant k$. For each i, we need to be able to compute the family $\mathcal{T}_i$ of admissible sets in polynomial time. This set family $\mathcal{T}_i$ is specific to the problem under consideration, but since it is used when computing Π, it must be computable in polynomial time. Then, we define F^{i+} as the set of all admissible sets $R \in \mathcal{T}_i$ such that $E_1 \cup \ldots \cup E_i \cup R \in \mathcal{F}$.

A crucial tool that was needed to achieve polynomial delay polynomial space enumeration for strongly accessible set systems [24] is the existence of a *prefix closed* ordering scheme. Due to the rigidity it implies, the prefix closed property helps to make sure to follow the "right path" when exploring the supergraph of solutions. This notion can easily be translated in the context of set-ordering schemes.

Definition 6.11 (Prefix closed)**.** *A set-ordering scheme is said to be* prefix closed *if it has the prefix property, and for any $F \in \mathcal{F}$ with $\Pi(F) = E_1, \ldots, E_k$ and for any $i < k$ there exists a preference relation $<_{F^i}$ over F^{i+} such that $E_{i+1} = \min_{<_{F^i}}\{X \in F^{i+} \mid X \subseteq F\}$.*

Remark that the so-called preference relation associated with a prefix F^i does not need to be a total ordering on F^{i+}. In fact, the important property is the existence of a unique minimal element in F^{i+} for $<_{F^i}$.

This means in particular that with a prefix closed ordering scheme, the next element is uniquely determined by the given prefix. Being prefix closed makes the (set-)ordering scheme easier to compute in polynomial time. Moreover, the existence of a preference relation can be used to derive an ordering on all solutions.

Total ordering on the solution set. For a prefix closed set-ordering scheme Π, it is possible to define a total ordering $\prec_\Pi$ over $\mathcal{F}_{\max}$ in the following way. Let $F_1, F_2 \in \mathcal{F}_{\max}$,

with $\Pi(F_1) = D_1, \ldots, D_k$, $\Pi(F_2) = E_1, \ldots, E_\ell$ and let $j \geqslant 0$ be the largest index such that the jth prefix of F_1 and F_2 is the same, that is, $F_1^j = D_1, \ldots, D_j = E_1, \ldots, E_j = F_2^j$. Let us denote by L the largest common prefix of F_1 and F_2, that is, $L := F_1^j = F_2^j$ (note that the set L is not necessarily equal to $F_1 \widetilde{\cap} F_2$, the largest prefix of F_2 contained in F_1, since its elements can appear in any order in F_1). Then $F_1 \prec_\Pi F_2$ if $D_{j+1} <_L E_{j+1}$.

From now on, we will assume that the problem under consideration admits a prefix closed ordering scheme. Therefore, our problem becomes the enumeration of all maximal elements of a set system $\mathcal{F}$, with a suitable, polynomial-time computable neighbouring function NEIGHBOURS and a prefix closed ordering scheme Π. We consider the functions PARENT and CHILDREN as defined previously.

We will see in what follows that being prefix closed is indeed the right notion to achieve polynomial space with canonical path reconstruction.

First, let us begin with a series of two technical lemmas. In general, nothing can be said about the canonical paths of two different solutions. However, if a solution F shares a common prefix with another solution on its canonical path, then it is possible to prove that they also share their canonical paths. It is the purpose of the following lemma.

Lemma 6.12. *Let $F \in \mathcal{F}_{\max}$ with $\Pi(F) = E_1, \ldots, E_\ell$; let $F_0, \ldots, F_k$ be its canonical path and let $i \leqslant k$. If $E_1, \ldots, E_{F_i \widetilde{\cap} F}$ is a prefix of $\Pi(F_i)$, then the canonical path of F_i is $F_0, \ldots, F_i$.*

Proof. Since $F_i \in \mathcal{F}_{\max}$, in particular it has a canonical path. Consider $T_0, \ldots, T_h$ the canonical path of F_i. First, by definition we have $T_0 = F_0$. Now, let $j < i$ be such that $T_0, \ldots, T_j = F_0, \ldots, F_j$. We will prove that $F_{j+1} = T_{j+1}$, implying by induction the equality $T_0, \ldots, T_i = F_0, \ldots, F_i$.

By Observation 6.7, since $j < i$ we know that $F_j \widetilde{\cap} F < F_i \widetilde{\cap} F$, so there exists $r < F_i \widetilde{\cap} F$ such that $E_1 \cup \ldots \cup E_r \subseteq F_j$ and $E_{r+1} \setminus F_j \neq \varnothing$. By hypothesis, we know that $E_1, \ldots, E_{r+1}$ is a prefix of $\Pi(F_i)$. Hence it follows $r = F_j \widetilde{\cap} F = F_j \widetilde{\cap} F_i < F_i \widetilde{\cap} F$.

Recall that Function NEXT is a deterministic function choosing among the neighbours of the current solution, one that increases the proximity to the target solution. In other words, NEXT puts a preference order $<_{\text{NEXT}}$ on the neighbours, and selects the minimum according to $<_{\text{NEXT}}$.

Using this notation, we can then write

$$\begin{aligned} F_{j+1} = \text{NEXT}(F_j, F) &= \min_{<_{\text{NEXT}}} \{F' \in \text{NEIGHBOURS}(F_j) \mid E_1 \cup \ldots \cup E_{r+1}\} \subseteq F'\} \\ &\quad \text{by definition of Function NEXT} \\ &= \min_{<_{\text{NEXT}}} \{F' \in \text{NEIGHBOURS}(T_j) \mid E_1 \cup \ldots \cup E_{r+1} \subseteq F'\} = \text{NEXT}(T_j, F_i) = T_{j+1} \\ &\quad \text{since } T_j = F_j \text{ and since } E_1, \ldots, E_{r+1} \text{ is a prefix of } F_i. \end{aligned}$$

Therefore, $F_0 = T_0$ and for any $j < i$ such that $F_j = T_j$, we have $F_{j+1} = T_{j+1}$. This implies in turn $T_1, \ldots, T_i = F_1, \ldots, F_i$ and the lemma is proven. □

The next technical lemma links the proximity between two solutions to the total order that was defined according to Π on the set of all solutions. This order will be crucial in the sequel, namely when proving that the function CHILDREN induces an acyclic relation on the solution space.

Lemma 6.13. *Let $F_1, F_2 \in \mathcal{F}$ with $\Pi(F_1) = D_1, \ldots, D_\ell$ and $\Pi(F_2) = E_1, \ldots, E_k$. Assume furthermore that there exists $j \leqslant \ell$ such that for any $i \leqslant j$, $D_i \subseteq F_2$. Then one of the two possibilities holds:*

1. *$F_2 \prec_\Pi F_1$;*
2. *$D_i = E_i$ for all $i \leqslant j$.*

Proof. Suppose that there exists $i \leqslant j$ such that $D_i \neq E_i$ (otherwise the second case holds), and let us consider the smallest such i. Then $F_1^{i-1} = F_2^{i-1}$. Denote by L this common prefix, that is, $L := F_1^{i-1} = F_2^{i-1}$. Moreover, we know that $D_i \subseteq F_2$ and $D_i \in L^+$. Since Π is prefix closed, it implies $E_i <_L D_i$, hence $F_2 \prec_\Pi F_1$. □

With the last two lemmas, we are now ready to prove the Function CHILDREN induces an arborescence on the solution set $\mathcal{F}_{\max}$, since we assumed that the ordering scheme Π is prefix closed.

Theorem 6.14. *If the ordering scheme Π is prefix closed, then* CHILDREN *defines a spanning arborescence on $\mathcal{F}_{\max}$ rooted at F_0.*

Proof. The supergraph of solutions defined by Function CHILDREN is the directed graph on $\mathcal{F}_{\max}$ with arc set $\{(F_i, F_j) \mid F_j \in \textsc{Children}(F_i)\}$. We will show that this supergraph of solutions is an arborescence.

Since each solution $F \in \mathcal{F}_{\max} \setminus \{F_0\}$ has only one in-neighbour $\textsc{Parent}(F)$, it is sufficient to show that for all $F \in \mathcal{F}_{\max}$ there exists a path from F_0 to F. Let us denote by $\mathcal{R}(F_0) \subseteq \mathcal{F}_{\max}$ the set of all sets $F \in \mathcal{F}_{\max}$ for which there exists a path from F_0 to F.

Assume for contradiction that $\mathcal{R}(F_0) \neq \mathcal{F}_{\max}$ and let $F := \min_{\prec_\Pi}(\mathcal{F}_{\max} \setminus \mathcal{R}(F_0))$. Let $\Pi(F) = E_1, \ldots, E_\ell$ and let $F_0, \ldots, F_k$ be the canonical path of F. Now, consider the smallest index $i^* \leqslant k$ such that $F_{i^*} \notin \mathcal{R}(F_0)$. If $i^* = k$ then $F_{k-1} \in \mathcal{R}(F_0)$ and since $F_{k-1} = \textsc{Parent}(F)$, F would belong to $\textsc{Children}(F_{k-1})$ and then F would belong to $\mathcal{R}(F_0)$. Hence, assume $i^* < k$.

Let $j := F_{i^*} \mathbin{\widetilde{\cap}} F$. By minimality of F with respect to $\prec_\Pi$, we have $F \prec_\Pi F_{i^*}$. Therefore, $F_{i^*} \prec_\Pi F$ does not hold (since $F_{i^*} \neq F$). Hence by Lemma 6.13, since $E_1 \cup \ldots \cup E_j \subseteq F_{i^*}$ by definition of the proximity, the jth prefix of $\Pi(F_{i^*})$ is $E_1, \ldots, E_j$. Now, by Lemma 6.12, the canonical path of F_{i^*} is $F_0, \ldots, F_{i^*}$, thus F_{i^*} has F_{i^*-1} as

its parent. By minimality of i^*, we know that $F_{i^*-1} \in \mathcal{R}(F_0)$ and since in addition $F_{i^*} \in \textsc{Children}(F_{i^*-1})$, we deduce $F_i^* \in \mathcal{R}(F_0)$, leading to a contradiction. □

It is somehow counter-intuitive to observe that the canonical path of a solution is not necessarily part of the final arborescence rooted at F_0. This canonical path is only computed to find the last solution in the path before F to define it as the parent of F. What we proved is that while the so-defined parent-child relation does not exactly follow the canonical paths, it still forms a spanning arborescence on the solution set rooted at F_0.

Now that we have an arborescence on the solution set, the idea is to explore it in a depth-first manner. The most suited algorithm is undoubtedly Reverse Search, explicitly designed for this purpose.

Observe that the naive implementation of Algorithm 2.2 (see Section 2.2.3) may use exponential space, since the height of the recursion tree might be exponential. Indeed,the recursion tree corresponds to the arborescence defined by the parent-child relation and we are not able to guarantee that its height is polynomial. However, since the functions Parent and Children are computable in polynomial time, we do not need to store the state of each recursion call. The classical trick introduced by Tsukiyama *et al.* in 1977 [84] and formalised by Avis & Fukuda in 1996 [2] as part of the Reverse Search algorithm is to use the function Parent to perform the backtrack operation on the fly. Indeed, this function is able to navigate backward in the tree, removing the need of keeping in memory all recursion calls. Thus, the algorithm only needs to keep in memory the current solution, and not the whole set of all the solutions that have already been found.

Therefore, Theorem 6.14 implies that, when the set-ordering scheme is both proximity searchable and prefix closed, it is possible to use the canonical path reconstruction method to enumerate $\mathcal{F}_{\max}$ efficiently. This rephrases as the following theorem.

Theorem 6.15. *Let $\mathcal{F}$ be a set system with a polynomially computable function* neighbours *and a prefix closed ordering scheme. If the enumeration of all maximal elements of $\mathcal{F}$ is proximity searchable, then $\mathcal{F}_{\max}$ can be enumerated with polynomial delay and polynomial space.*

6.3 Applications of canonical path reconstruction

In the present section, we take interest on some problems for which canonical path reconstruction provides an efficient, polynomial space algorithm. This is for example the case for minimal chordal completions, whose ordering scheme is naturally prefix closed. However, it can be successfully applied to other problems. For example, a polynomial delay algorithm was known for the enumeration of maximal induced chordal subgraphs

[23], but this algorithm runs with exponential space. We exhibit here a prefix closed ordering scheme for this problem, allowing us to use canonical path reconstruction, therefore reducing the space complexity of the maximal induced chordal subgraphs enumeration problem.

6.3.1 Minimal chordal completions in polynomial space

In Chapter 5, we proved that minimal chordal completions are proximity searchable with the canonical ordering Can (see page 93), thus there exists a polynomial delay algorithm for their enumeration. We would like to make the enumeration process work in polynomial space, since this could lead to an algorithm that is more usable in practice. To achieve this space complexity, a first idea would be to use the Proximity Search technique developed for commutable set systems [24], when the ordering is prefix closed.

As stated earlier, chordal completions form a strongly accessible set system since they are sandwich-monotone, and so are their complements. However, the set of chordal completions is not a commutable set system, and neither is the set of their complements. Take for example the completion H of a P_4 containing an edge e joining the two endpoints, and the two chords a and b, as shown in Figure 6.4. The P_4 given by $H \setminus \{a, b, e\}$ is chordal. Plus, both graphs $H \setminus \{a\}$ and $H \setminus \{b\}$ are chordal, whereas $H \setminus \{a, b\}$ is not.

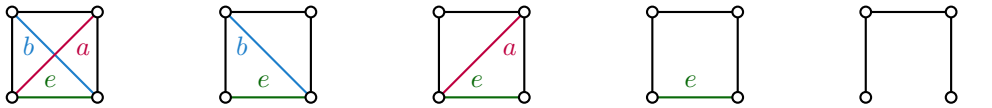

Figure 6.4: Chordal completions of an arbitrary graph (here a P_4) are not a commutable set system.

Therefore, we cannot try to enumerate the minimal chordal completions of a graph in polynomial space with this technique. Instead, we will prove the problem satisfies the conditions of Theorem 6.15. With canonical path reconstruction, it will be possible to define a suitable parent-child relation in order to organise the space of solutions in an arborescence. This way, the enumeration of minimal chordal completions of a graph will need only polynomial space.

Neighbouring function and ordering scheme. For this enumeration problem, we use the same neighbouring function (based on the Flip operation) and ordering scheme (namely, Can) as in Section 5.3. It is stated in Theorem 5.20 that the minimal chordal completions enumeration problem, with the corresponding neighbouring function and the ordering scheme Can, is proximity searchable.

To guarantee we are in the conditions of Theorem 6.15, it remains only to prove that the ordering scheme Can is prefix closed. This will not be very difficult since Can has been built this way.

Lemma 6.16. *The ordering scheme* Can *defined in Section 5.3.1 is prefix closed.*

Proof. By construction, Can uses a preference relation on the non-elements of a partial solution, that is just the arbitrary ordering of E^c among the candidates. Moreover, since the chordal completions of a graph are sandwich-monotone, it follows immediately that the ordering scheme Can has the prefix property.

Therefore, Can is straightforwardly prefix closed. □

Now, we are sure all requirements of Theorem 6.15 are satisfied. Therefore, it is possible to define for each solution $F \neq F_0$ its parent $\textsc{Parent}(F)$ as the last solution before F in its canonical path, as in Section 6.2. The parent of a solution is computable in polynomial time by lemma 6.9. We deduce the following result.

Theorem 6.17. *The minimal chordal completions of a graph can be enumerated with polynomial delay in polynomial space.*

6.3.2 Maximal induced chordal subgraphs

In this section, we show that the canonical path reconstruction framework can be used on other problems that were known to admit a polynomial delay algorithm, but for which the existence of such an algorithm running in polynomial space was unknown. This is for example the case with two enumeration problems concerning chordal subgraphs: the enumeration of maximal induced chordal subgraphs and maximal *connected* induced chordal subgraphs. Both problems have been studied by Conte *et al.* in the light of the Proximity Search framework. Unless stated otherwise, any mention of their work in this section refers in fact to the original Proximity Search paper [23].

Here we consider the problem of listing maximal induced chordal subgraphs of a graph G. In the following, we assume that vertices of G are arbitrarily ordered. Since we work with induced subgraphs, the solutions will be seen as sets of vertices of G.

It is possible to enumerate maximal induced chordal subgraphs with polynomial delay, using the classical Proximity Search [23]. However, the algorithm obtained this way uses exponential space. We will see that the canonical path reconstruction framework can be used to reduce the space complexity of the problem.

Neighbouring function. The neighbouring function used by Conte *et al.*, that we will use as well, is defined as follows. For a solution $S \subset V$, choose a vertex v of $V \setminus S$, and choose a maximal clique Q in $N_G(v) \cap G[S]$. Then, add v to the solution and remove all neighbours of v in S that are not in Q (that is, remove $N_G(v) \setminus Q$

from the solution). This yields an induced chordal subgraph of G. It remains only to complete greedily the newly obtained solution into a maximal one denoted $S_{v,Q}$. The neighbouring function for induced chordal subgraphs is illustrated in Figure 6.5.

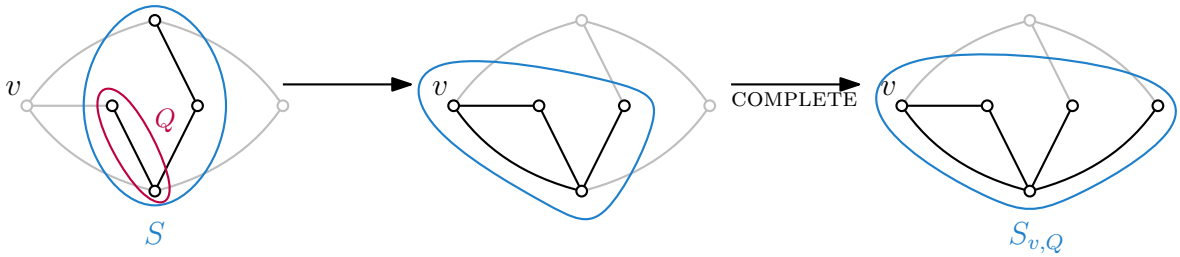

Figure 6.5: Illustration of the neighbouring function for induced chordal subgraphs.

With the neighbouring function just defined, the set of neighbours of the solution S is defined as $\text{NEIGHBOURS}(S) := \{S_{v,Q} \mid v \notin S,\ Q \text{ maximal clique in } N_G(v) \cap S\}$. Since S is chordal, it has a linear number of cliques [46, 49], so there are a polynomial number of choices for Q. Therefore each solution has a polynomial number of neighbours, and the authors proved that this neighbouring function is proximity searchable.

Ordering scheme. It remains to find a suitable ordering on the elements of a solution. Recall that a *perfect elimination ordering* (PEO, defined page 19) of a graph is an ordering $v_1, \ldots, v_n$ of the vertices such that at each step i, the vertex v_i is simplicial (its neighbourhood is a clique) in the subgraph induced by $\{v_i, \ldots, v_n\}$, and chordal graphs are characterised by the existence of a PEO. In the original Proximity Search paper, the authors proved that all ordering schemes that produce reverse perfect elimination orderings of solutions are proximity searchable. We exhibit here such an ordering scheme which, in addition, is prefix closed.

Indeed, the ordering scheme that was previously used to prove the existence of a polynomial delay algorithm for maximal induced chordal subgraphs fails to be prefix closed. More precisely, this ordering scheme consists in computing the entire lexicographically minimal perfect elimination ordering of the solution before reversing it. Consequently, if S is a maximal induced chordal subgraph with its reverse lexicographically minimal PEO, the ordering $<_{S_i}$ of its ith prefix is not well defined in general. On the other hand, the "natural" greedy ordering, that can easily be computed on any solution, is prefix closed by definition but fails to be proximity searchable.

If $G' = G[S]$ is an induced chordal subgraph of G, we consider $\pi(G') = x_1, \ldots, x_k$ the ordering of G' such that $x_1 = \min(S)$ and for each $i < k$, x_{i+1} is the smallest among vertices of maximum degree in $\{x_1, \ldots, x_i\}$. In other words, we select at each step a vertex which has a maximum number of neighbours in the set of vertices already chosen. It has been proven by Tarjan & Yannakakis [82] that this procedure known as

Maximum Cardinality Search (MCS for short), and presented here as Algorithm 6.1, produces a reverse perfect elimination ordering whenever G' is chordal.

Algorithm 6.1: Maximum Cardinality Search (MCS)[82]

input : A graph $G = (V, E)$
output : A reverse PEO of G if G is chordal

```
1 Begin with all vertices unnumbered.
2 foreach vertex v ∈ V do
3     weight(v) := 0;
4 for i = 1 to n do
5     Find the unnumbered vertex u of maximum weight (break ties with smallest
        vertex id);
6     v_i := u;
7     foreach unnumbered v ∈ N_G(v_i) do
8         weight(v) := weight(v) + 1;
9 return v_1, ..., v_n;
```

To illustrate how Algorithm 6.1 works, let us have a look on the example of Figure 6.6. The graph is chordal and has vertices a, b, c, d, e, and f. At first, the algorithm selects the vertex of smallest id, a. Then, weights are put on the neighbours of a, symbolised by dots on the picture. At each step, the algorithm selects the vertex of largest weight and increases the weights of its neighbours, until all vertices have been eliminated. In this example, the graph is chordal and the ordering returned by Algorithm 6.1 is $abedcf$, which is indeed the reverse of a PEO.

Since the ordering scheme produced by Maximum Cardinality Search, called here *MCS-ordering scheme*, is the reverse of a perfect elimination ordering on chordal graphs, it follows [23] that our problem is proximity searchable. This rephrases as the following lemma.

Lemma 6.18. *The problem of enumerating maximal induced chordal subgraphs with the previously defined neighbouring function and the MCS-ordering scheme is proximity searchable.*

In order to apply Theorem 6.15, we need our ordering scheme to be prefix closed. This will allow us to use canonical path reconstruction with the guarantee that every solution is outputted exactly once, therefore using only polynomial space. That is what we will do when proving the following lemma.

Lemma 6.19. *The MCS-ordering scheme is prefix closed.*

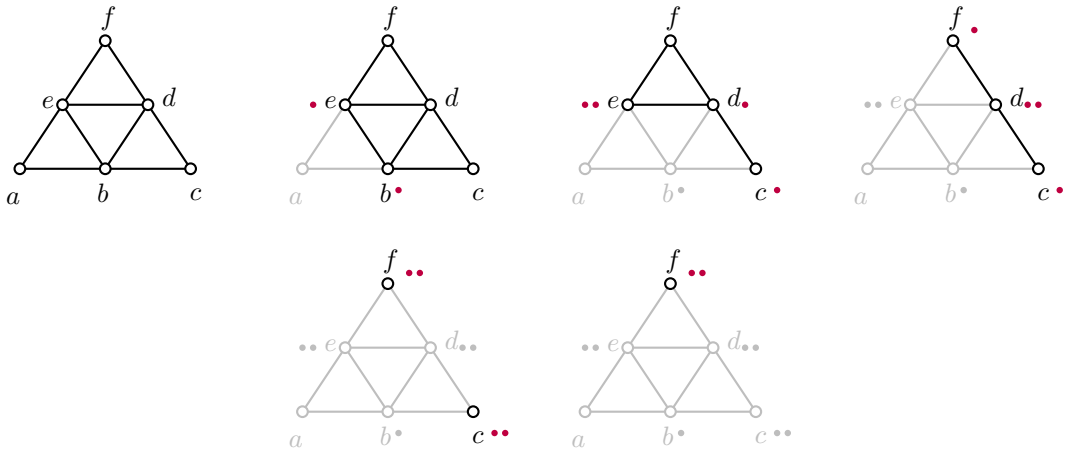

Figure 6.6: Example of an execution of Algorithm 6.1 on a chordal graph. The weights on the vertices are symbolised by dots.

Proof. Let G be a graph. A maximal chordal subgraph S_0 of G can be found by applying a greedy algorithm adding vertices one by one while the resulting induced subgraph is chordal.

First, the MCS-ordering scheme has the prefix property. Indeed, the property of being chordal is hereditary, therefore any prefix of the canonical ordering of a solution is an induced subgraph of this solution, that is, a chordal graph.

Then, since the property of being chordal is hereditary, in particular it is also strongly accessible, and the set of candidates X^+ for any partial solution X is well-defined and non-empty unless X is a maximal induced chordal subgraph of G. Let us consider a (non-maximal) set of vertices X which induces a chordal subgraph of G. By definition, the elements of X^+ are the vertices $x \in V(G) \setminus X$ such that $G[X \cup \{x\}]$ is chordal.

We give an ordering $<_X$ of X^+ in the following manner. We say that $x <_X y$ if either $|N(x) \cap X| > |N(y) \cap X|$ or $N(x) \cap X = N(y) \cap X$ and $x < y$. This way, vertices of large degree in X are ordered first and the ties are broken according to the arbitrary ordering of the vertices. In other words, the vertices of X^+ are ordered by considering the lexicographic order of the tuple $(n - |N(x) \cap X|, x)$, with $x \in X^+$.

Equipped with this ordering of X^+ for any subset of vertices inducing a chordal subgraph, it is straightforward to see that the MCS-ordering scheme satisfies Definition 6.11. □

We are now able to use Proximity Search on the set of maximal induced chordal subgraphs of G, and ensure that the ordering we consider is prefix closed. Therefore our setting satisfies the requirements for Theorem 6.15. Hence, for each solution S different from the root solution S_0, it is possible to compute in polynomial time the canonical path of S. The parent $\textsc{Parent}(S)$ of S is then defined as the last solution before S on its canonical path, as in Section 6.2. This implies the following result.

Theorem 6.20. *Maximal induced chordal subgraphs can be enumerated with polynomial delay in polynomial space.*

6.3.3 Maximal connected induced chordal subgraphs

In fact, the MCS-ordering scheme is also prefix closed when considering maximal *connected* induced chordal subgraphs. It has been proved by Conte *et al.* that – with the appropriate neighbouring function they provide – the problem of enumerating maximal connected induced chordal subgraphs with any ordering scheme corresponding to the reverse of a perfect elimination ordering is proximity searchable. Moreover, by definition the MCS-ordering scheme also has the prefix property for connected induced chordal subgraphs. Therefore, the proof of Lemma 6.19 also holds in the case of connected induced chordal subgraphs. As a consequence, the same result holds for maximal connected induced chordal subgraphs.

Theorem 6.21. *Maximal connected induced chordal subgraphs can be enumerated with polynomial delay in polynomial space.*

Conclusion

The ultimate goal of a thesis on the subject would be to find a general technique, applicable to the most possible graph classes, for the enumeration of minimal completions, minimal deletions, maximal induced subgraphs that belong to the considered class. If such a general technique does not exist for all (hereditary) classes of graphs, then a natural question arises: which graph classes admit an efficient (*e.g.* polynomial delay) enumeration algorithm for minimal completions (resp. deletions) or maximal induced subgraphs, and which ones do not?

Neither of these questions is an easy task. In this thesis, advances have been made concerning the theoretical knowledge of the minimal completions, minimal deletions, and maximal induced subgraphs enumeration problems.

Many efficient techniques already exist for enumeration purposes, and they are often well suited to the problems we considered here, in particular to maximal induced subgraphs enumeration. For some graph classes, such as split graphs, cographs, and threshold graphs, the existence of a polynomial delay algorithm enumerating maximal induced subgraphs was still unknown. We showed that these three problems could be solved efficiently using existing techniques, by relying on the structural characterisations of graphs in those three classes.

For split graphs, the input restricted problem introduced by Cohen, Kimelfeld & Sagiv [22] proved its efficiency regarding both time and space. For the enumeration of maximal induced sub-cographs and threshold subgraphs, we managed to apply the more recent *Proximity Search* technique proposed by Conte & Uno [26], and exhibited non-trivial polynomial delay, exponential space algorithms for these two problems.

When it comes to minimal completions and deletions, it seems even harder in general to find a suitable way to apply an existing technique. However, in certain cases, the properties satisfied by the graphs in the considered class allow us to link the minimal completions with other structures for which the complexity of enumeration is known. It is the case with minimal split completions: due to the many properties satisfied by split graphs, it is in fact equivalent to enumerate the (inclusion-wise) maximal stable sets of a modified graph. Fortunately, the last problem has been known to be

solvable with polynomial delay in polynomial space for a long time [84]. An efficient algorithm for the enumeration of minimal split completions (and deletions since the class is self-complementary) follows immediately. Nevertheless, there is obviously no reason for such a correspondence to be found for any class of graphs.

In general, the assumptions for using an existing enumeration technique that yields good complexity results (Reverse Search, input restricted problem) are quite restrictive. Despite Flashlight Search to achieve good complexity results very simply, we showed in Chapter 4 that it cannot be used as a general technique for the enumeration of maximal induced subgraphs in a prescribed class $\mathcal{P}$. Indeed, the associated extension problem is NP-hard when no further assumption is made on $\mathcal{P}$.

Therefore, other options have to be considered: newer, less known techniques seem promising. We have seen in particular in Chapter 5 that Proximity Search turns up useful when enumerating maximal induced sub-cographs and threshold subgraphs. In fact, Proximity Search can also be used for minimal deletions enumeration problems. For example, it allows us to enumerate all minimal threshold deletions (and completions) with polynomial delay, whereas the class of threshold graphs is not monotone – the counterpart of hereditary for edge problems.

Although threshold graphs are not monotone, they still possess an interesting property: they are sandwich-monotone. This property makes the computation of a minimal deletion and the completion process into a minimal deletion easier when applying Proximity Search. However, it is quite a strong requirement, and it is not satisfied by all graph classes: for example cographs, trivially perfect graphs, P_3-free graphs, are not sandwich-monotone.

Polynomial sandwich algorithms [50] have been briefly mentioned in this thesis: when they exist – it is the case for cographs, trivially perfect graphs, and P_3-free graphs [50][10, $v1$] –, a polynomial-time computable completion function can be found even if the class is not sandwich-monotone. This could be useful to apply Proximity Search to many new problems. We leave as an open question the minimal deletions enumeration problems into the three aforementioned classes. Alongside that, further investigation on sandwich problems appears to be another interesting path to follow for future research.

Question 4. *Can minimal deletions of a graph into cographs, trivially perfect graphs, or P_3-free graphs be enumerated with polynomial delay (with or without the help of Proximity Search)?*

Finally, the question of how efficiently it is possible to enumerate the minimal chordal completions of a graph has been a hot topic in research for at least the last decade [18, 8]. The best algorithm up to date was proposed in 2017 and has incremental delay. We were interested in this problem and we proved Proximity Search can also be used to enumerate the minimal chordal completions with polynomial delay. Our

approach consisted in considering the complements of the chordal completions to define the proximity measure, which – to the best of our knowledge – had never been done previously.

Then, to be useful in practice, an efficient algorithm should only use polynomial space. That is why we considered the question of space usage for minimal chordal completions. Thanks to the sandwich monotonicity of chordal graphs, we were able to prove the existence of a polynomial delay, polynomial space algorithm. For this purpose, we introduced a parent relation between two solutions, permitting to explore the solution space without keeping in memory all the solutions already found.

Generalising this approach gave rise to a new extension of Proximity Search to obtain polynomial space: *canonical path reconstruction*. The requirements of this technique are weaker than those of the other Proximity Search-based polynomial space enumeration techniques [23]. Consequently, our framework can be applied to a wider range of problems, guaranteeing polynomial delay and polynomial space under weaker assumptions. Some of the problems that were solved at the beginning with Proximity Search satisfy in fact those requirements, one of which is the class being sandwich-monotone. For example, our technique directly gives an algorithm listing the maximal induced (connected or not) chordal subgraphs of a graph in polynomial space.

The results about polynomial delay enumeration algorithms mentioned in this thesis are summarised in Table 7.1.

Graph class $\mathcal{P}$	Maximal induced subgraphs	Minimal $\mathcal{P}$-deletions	Minimal $\mathcal{P}$-completions
Split graphs	PDelay, PSPACE Cao [13] / Theorem 3.17 *input restricted problem* Section 3.2	PDelay, PSPACE Theorem 3.10 *self-complementarity* Section 3.1	PDelay, PSPACE Theorem 3.10 *ad hoc bijection* Section 3.1
Cographs	PDelay Theorem 5.4 *Proximity Search* Section 5.1	OPEN	OPEN
Threshold graphs	PDelay, PSPACE Cao [13] *input restricted problem*	PDelay Theorem 5.12 *Proximity Search* Section 5.2.2	PDelay Theorem 5.12 *self-complementarity* Section 5.2.2
Chordal graphs	PDelay, PSPACE Conte *et al.* [23] & Theorem 6.20 *canonical path reconstruction* Section 6.3.2	PDelay Conte *et al.* [23] *Proximity Search*	PDelay, PSPACE Theorem 6.17 *canonical path reconstruction* Sections 5.3 and 6.3.1

Table 7.1: Known complexities of the maximal induced subgraphs, minimal completions and deletions enumeration problems studied in this thesis. PDelay stands for "polynomial delay" and PSPACE for "polynomial space".

When considering the maximal subgraphs and minimal supergraphs, other classes of interest include – but are not limited to – triangle-free graphs (which may be a candidate for the non-existence of a polynomial delay algorithm), perfect graphs, and planar

graphs. To the best of our knowledge, no polynomial delay algorithm for maximal induced subgraphs is known in any of those classes.

In fact, all the Proximity Search-based techniques currently used rely heavily on solution orderings. We proved that it is possible to relax the notion of ordering scheme, introducing *set-ordering schemes*. With set-ordering schemes, solutions are not seen as ordered subsets of the ground set any more, but rather as ordered partitions. This extension of the "usual" Proximity Search technique could help to find some new problems to which Proximity Search can be applied, implying the existence of a polynomial delay algorithm. If moreover the problems satisfy the requirements for canonical path reconstruction, then polynomial space will be achieved as well.

General set proximity makes it natural to think about some minimal deletions problems as derived from the corresponding maximal induced subgraphs problems. This was for example the case when Proximity Search was applied to enumerate the minimal threshold deletions of a graph. Naturally, the following question arises.

Question 5. *In which cases can an algorithm enumerating the minimal deletions be derived from the maximal induced subgraphs enumeration with the help of general set proximity?*

Minimal completions and deletions are not the only interesting objects that can be enumerated in graphs. *Potential Maximal Cliques* of a graph were introduced in the early 2000s [9, 83] as the sets of vertices inducing a clique in some minimal chordal completion. Due to their links with chordal completions and minimal separators, they are a powerful tool for the computation of treewidth and minimum fill-in. Currently, the best algorithm known for their enumeration [9] is quadratic in the number of solutions and uses exponential space. For applications, it would be interesting to be able to generate them more efficiently. This brings us to a problem that has been open for twenty years.

Question 6. *Is it possible to generate all Potential Maximal Cliques of a given graph with polynomial delay, or in polynomial space?*

Contributions

Contributions on enumeration

The results presented in this thesis gave rise to the following two articles, one of which is published [11] and the other one is submitted to a specialised journal [10].

- **Efficient enumeration of maximal split subgraphs and sub-cographs and related classes**, with Aurélie Lagoutte, Vincent Limouzy, Arnaud Mary and Lucas Pastor, preprint on *ArXiv* arXiv:2007.01031, 2020. Submitted to *Discrete Applied Mathematics*.
- **Polynomial delay algorithm for minimal chordal completions**, with Vincent Limouzy and Arnaud Mary, in *49th International Colloquium on Automata, Languages, and Programming (ICALP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

Contributions on other subjects

Other results have been obtained during my PhD, that have not been presented in the present manuscript but gave rise to two papers, submitted to or published in specialised journals. They are about graph theory but do not concern enumeration.

Connected greedy colourings First, a project took place with the AlCoLoCo working group in Clermont-Ferrand about connected greedy colourings of graphs.

Given a graph G and a vertex ordering $\sigma : v_1 \dots v_n$ of $V(G)$, the "first-fit" greedy colouring algorithm gives colours to all vertices of G in order σ by assigning to each vertex the smallest colour unused in its neighbourhood. By restricting this procedure to vertex orderings that are connected, we obtain *connected greedy colourings*.

For some graphs, connected greedy colourings use at most $\chi(G)$ colours (that is, the minimum feasible number of colours). However, it is not always the case: sometimes there does not exist any connected greedy colouring using less than $\chi(G)+1$ colours. A graph G with this last property is called *ugly*. The question we worked on was the following: for a given graph class, does it contain ugly graphs? We partially answered it for perfect graphs and K_4-minor-free graphs, proving that no perfect graph is ugly, and no K_4-minor-free graph is ugly.

These results appear in the preprint [4]

- **Connected greedy colourings of perfect graphs and other classes: the good, the bad and the ugly**, with Laurent Beaudou, Oscar Defrain, Florent Foucaud, Aurélie Lagoutte, Vincent Limouzy and Lucas Pastor, on *ArXiv* arXiv:2110.14003, 2021. Submitted for publication.

Locating-dominating sets in oriented graphs The other project was realised with a team constituted during the annual French graph meeting JGA 2020.

Given an oriented graph $D=(V,A)$ on n vertices, a *dominating set* is a set S of vertices of D such that any vertex in $V\setminus S$ has at least one in-neighbour in S. A set L is *locating* if for any two vertices u, v in $V\setminus L$, the in-neighbourhoods of u and v are different sets – permitting to differentiate u and v. We were interested in knowing the maximum size of a minimum set that is both locating and dominating.

It was already known [44] that for tournaments, that is, orientations of complete graphs, such a set can contain at most $\lceil n/2\rceil$ vertices. We extended this result to a larger class of oriented graphs. An oriented graph is a *local tournament* if the in- and out-neighbourhoods of any vertex induce tournaments. It is of course the case for tournaments, but this class contains way more oriented graphs; for example, oriented cycles are local tournaments.

We proved that any connected local tournament on n vertices has a locating-dominating set of size at most $\lceil n/2\rceil$.

We also proved a conjecture of Foucaud, Heydarshahi & Parreau [44] in a special case: when the oriented graph has no twins nor quasi-twins – for oriented graphs, two vertices sharing the same open (resp. closed) in-neighbourhood –, and has a vertex from which any other can be reached. In this case, we proved that the oriented graph has a locating-dominating set of size at most $\frac{2n}{3}$, and that this bound is tight, meeting the value that was previously conjectured.

These results can be found in the article [5]

- **Locating-dominating sets in local tournaments**, with Thomas Bellitto, Benjamin Lévêque and Aline Parreau, *Discrete Applied Mathematics*, 337:14–24, 2023.

Bibliography

[1] Sophie Achard, Chantal Delon-Martin, Petra E. Vértes, Félix Renard, Maleka Schenck, Francis Schneider, Christian Heinrich, Stéphane Kremer, and Edward T. Bullmore. Hubs of brain functional networks are radically reorganized in comatose patients. *Proceedings of the National Academy of Sciences*, 109(50):20608–20613, 2012.

[2] David Avis and Komei Fukuda. Reverse search for enumeration. *Discrete Applied Mathematics*, 65(1-3):21–46, 1996.

[3] Guillaume Bagan. *Algorithms and complexity of enumeration problems for the evaluation of logical queries*. PhD thesis, University of Caen Normandy, France, 2009. https://theses.hal.science/tel-00424232.

[4] Laurent Beaudou, Caroline Brosse, Oscar Defrain, Florent Foucaud, Aurélie Lagoutte, Vincent Limouzy, and Lucas Pastor. Connected greedy colourings of perfect graphs and other classes: the good, the bad and the ugly. *arXiv preprint arXiv:2110.14003*, 2021. https://arxiv.org/abs/2110.14003.

[5] Thomas Bellitto, Caroline Brosse, Benjamin Lévêque, and Aline Parreau. Locating-dominating sets in local tournaments. *Discrete Applied Mathematics*, 337:14–24, 2023.

[6] Jan C. Bioch and Toshihide Ibaraki. Complexity of identification and dualization of positive boolean functions. *Information and Computation*, 123(1):50–63, 1995.

[7] Endre Boros, Vladimir Gurvich, Khaled Elbassioni, and Leonid Khachiyan. An efficient incremental algorithm for generating all maximal independent sets in hypergraphs of bounded dimension. *Parallel Processing Letters*, 10(04):253–266, 2000.

[8] Endre Boros, Benny Kimelfeld, Reinhard Pichler, and Nicole Schweikardt. Enumeration in data management (dagstuhl seminar 19211). Technical report,

Dagstuhl Seminar, 2019. https://drops.dagstuhl.de/opus/volltexte/2019/11382/pdf/dagrep_v009_i005_p089_19211.pdf.

[9] Vincent Bouchitté and Ioan Todinca. Listing all potential maximal cliques of a graph. *Theoretical Computer Science*, 276(1-2):17–32, 2002.

[10] Caroline Brosse, Aurélie Lagoutte, Vincent Limouzy, Arnaud Mary, and Lucas Pastor. Efficient enumeration of maximal split subgraphs and induced sub-cographs and related classes. *arXiv preprint arXiv:2007.01031*, 2020. https://arxiv.org/abs/2007.01031.

[11] Caroline Brosse, Vincent Limouzy, and Arnaud Mary. Polynomial delay algorithm for minimal chordal completions. In *49th International Colloquium on Automata, Languages, and Programming (ICALP 2022). Schloss Dagstuhl-Leibniz-Zentrum für Informatik*, 2022.

[12] Tiziana Calamoneri, Mattia Gastaldello, Arnaud Mary, Marie-France Sagot, and Blerina Sinaimeri. On maximal chain subgraphs and covers of bipartite graphs. In Vittorio Bilò and Antonio Caruso, editors, *Proceedings of the 17th Italian Conference on Theoretical Computer Science, Lecce, Italy, September 7-9, 2016*, volume 1720 of *CEUR Workshop Proceedings*, pages 286–291. CEUR-WS.org, 2016.

[13] Yixin Cao. Enumerating maximal induced subgraphs. *arXiv preprint arXiv:2004.09885*, 2020. https://arxiv.org/abs/2004.09885.

[14] Florent Capelli and Yann Strozecki. Incremental delay enumeration: Space and time. *Discrete Applied Mathematics*, 268:179–190, 2019.

[15] Florent Capelli and Yann Strozecki. Enumerating models of DNF faster: breaking the dependency on the formula size. *Discrete Applied Mathematics*, 303:203–215, 2021.

[16] Florent Capelli and Yann Strozecki. Geometric amortization of enumeration algorithms. *arXiv preprint arXiv:2108.10208*, 2021. https://arxiv.org/abs/2108.10208.

[17] Nofar Carmeli, Batya Kenig, and Benny Kimelfeld. Efficiently enumerating minimal triangulations. In Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts, editors, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 273–287. ACM, 2017.

[18] Nofar Carmeli, Batya Kenig, Benny Kimelfeld, and Markus Kröll. Efficiently enumerating minimal triangulations. *Discrete Applied Mathematics*, In Press.

[19] Nofar Carmeli, Shai Zeevi, Christoph Berkholz, Benny Kimelfeld, and Nicole Schweikardt. Answering (unions of) conjunctive queries using random access and random-order enumeration. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 393–409, 2020.

[20] Katrin Casel, Henning Fernau, Mehdi Khosravian Ghadikolaei, Jérôme Monnot, and Florian Sikora. On the complexity of solution extension of optimization problems. *Theoretical Computer Science*, 904:48–65, 2022.

[21] Václav Chvátal and Peter L. Hammer. Aggregations of inequalities. *Studies in Integer Programming, Annals of Discrete Mathematics*, 1:145–162, 1977.

[22] Sara Cohen, Benny Kimelfeld, and Yehoshua Sagiv. Generating all maximal induced subgraphs for hereditary and connected-hereditary graph properties. *Journal of Computer and System Sciences*, 74(7):1147–1159, 2008.

[23] Alessio Conte, Roberto Grossi, Andrea Marino, Takeaki Uno, and Luca Versari. Proximity search for maximal subgraph enumeration. *SIAM Journal on Computing*, 51(5):1580–1625, 2022.

[24] Alessio Conte, Roberto Grossi, Andrea Marino, and Luca Versari. Listing maximal subgraphs satisfying strongly accessible properties. *SIAM J. Discrete Math.*, 33(2):587–613, 2019.

[25] Alessio Conte, Roberto Grossi, Giulia Punzi, and Takeaki Uno. Enumeration of Maximal Common Subsequences Between Two Strings. *Algorithmica*, 84(3):757–783, 2022.

[26] Alessio Conte and Takeaki Uno. New polynomial delay bounds for maximal subgraph enumeration by proximity search. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019.*, pages 1179–1190, 2019.

[27] Irene Córdoba, Gherardo Varando, Concha Bielza, and Pedro Larrañaga. On generating random Gaussian graphical models. *International Journal of Approximate Reasoning*, 125:240–250, 2020.

[28] Derek G. Corneil, Helmut Lerchs, and L. Stewart Burlingham. Complement reducible graphs. *Discrete Applied Mathematics*, 3(3):163–174, 1981.

[29] Derek G. Corneil, Yehoshua Perl, and Lorna K. Stewart. A linear recognition algorithm for cographs. *SIAM Journal on Computing*, 14(4):926–934, 1985.

[30] Jean-François Couturier, Pinar Heggernes, Pim Van't Hof, and Dieter Kratsch. Minimal dominating sets in graph classes: combinatorial bounds and enumeration. *Theoretical Computer Science*, 487:82–94, 2013.

[31] Nadia Creignou, Markus Kröll, Reinhard Pichler, Sebastian Skritek, and Heribert Vollmer. A complexity theory for hard enumeration problems. *Discrete Applied Mathematics*, 268:191–209, 2019.

[32] Christophe Crespelle. Linear-time minimal cograph editing. In *International Symposium on Fundamentals of Computation Theory*, pages 176–189. Springer, 2021.

[33] Christophe Crespelle, Daniel Lokshtanov, Thi Ha Duong Phan, and Eric Thierry. Faster and enhanced inclusion-minimal cograph completion. In *International Conference on Combinatorial Optimization and Applications*, pages 210–224. Springer, 2017.

[34] Clément Dallard, Martin Milanič, and Kenny Štorgel. Treewidth versus clique number. II. Tree-independence number. *arXiv preprint arXiv:2111.04543*, 2021. https://arxiv.org/abs/2111.04543.

[35] Oscar Defrain. *On the dualization problem in graphs, hypergraphs, and lattices*. PhD thesis, Université Clermont Auvergne (2017-2020), 2020. https://theses.hal.science/tel-03036782v1.

[36] Reinhard Diestel. Graph theory. *Graduate Texts in Mathematics*, Springer-Verlag, 2005.

[37] G. A. Dirac. On rigid circuit graphs. *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, 25:71–76, 1961.

[38] Arnaud Durand. Workshop on graphs and databases, oral communication, Lyon, 2023.

[39] Richard Durstenfeld. Algorithm 235: random permutation. *Communications of the ACM*, 7(7):420, 1964.

[40] Eduard Eiben, William Lochet, and Saket Saurabh. A polynomial kernel for paw-free editing. *arXiv preprint arXiv:1911.03683*, 2019. https://arxiv.org/abs/1911.03683.

[41] Thomas Eiter and Georg Gottlob. Identifying the minimal transversals of a hypergraph and related problems. *SIAM J. Comput.*, 24(6):1278–1304, 1995.

[42] Stéphane Foldes and Peter L. Hammer. *Split graphs*. Universität Bonn. Institut für Ökonometrie und Operations Research, 1976.

[43] Stéphane Foldes and Peter L. Hammer. Split graphs having dilworth number two. *Canadian Journal of Mathematics*, 29(3):666–672, 1977.

[44] Florent Foucaud, Shahrzad Heydarshahi, and Aline Parreau. Domination and location in twin-free digraphs. *Discrete Applied Mathematics*, 284:42–52, 2020.

[45] Michael L. Fredman and Leonid Khachiyan. On the complexity of dualization of monotone disjunctive normal forms. *J. Algorithms*, 21(3):618–628, 1996.

[46] Delbert Fulkerson and Oliver Gross. Incidence matrices and interval graphs. *Pacific Journal of Mathematics*, 15(3):835–855, 1965.

[47] Michael R. Garey and David S. Johnson. *Computers and intractability: a guide to the theory of np-completeness*. San Francisco, California: W. H. Freeman and Co., 1979.

[48] Fănică Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. *Journal of Combinatorial Theory, Series B*, 16(1):47–56, 1974.

[49] Martin Charles Golumbic. *Algorithmic graph theory and perfect graphs*, volume 57. Elsevier, 2004.

[50] Martin Charles Golumbic, Haim Kaplan, and Ron Shamir. Graph sandwich problems. *Journal of Algorithms*, 19(3):449–473, 1995.

[51] Michel Habib and Christophe Paul. A simple linear time algorithm for cograph recognition. *Discrete Applied Mathematics*, 145(2):183–197, 2005.

[52] Rudolf Halin. S-functions for graphs. *Journal of Geometry*, 8:171–186, 1976.

[53] Peter L. Hammer and Bruno Simeone. The splittance of a graph. *Combinatorica*, 1:275–284, 1981.

[54] Pinar Heggernes. Minimal triangulations of graphs: a survey. *Discrete Mathematics*, 306(3):297–317, 2006.

[55] Pinar Heggernes and Federico Mancini. Minimal split completions. *Discrete Applied Mathematics*, 157(12):2659–2669, 2009.

[56] Pinar Heggernes and Charis Papadopoulos. Single-edge monotonic sequences of graphs and linear-time algorithms for minimal completions and deletions. *Theoretical Computer Science*, 410(1):1–15, 2009.

[57] David S. Johnson, Mihalis Yannakakis, and Christos H. Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27(3):119–123, 1988.

[58] Selmer M. Johnson. Generation of permutations by adjacent transposition. *Mathematics of computation*, 17(83):282–285, 1963.

[59] Heinz A Jung. On a class of posets and the corresponding comparability graphs. *Journal of Combinatorial Theory, Series B*, 24(2):125–133, 1978.

[60] Mamadou Moustapha Kanté, Vincent Limouzy, Arnaud Mary, and Lhouari Nourine. On the enumeration of minimal dominating sets and related notions. *SIAM J. Discret. Math.*, 28(4):1916–1929, 2014.

[61] Leonid Khachiyan, Endre Boros, Khaled Elbassioni, and Vladimir Gurvich. On the dualization of hypergraphs with bounded edge-intersections and other related classes of hypergraphs. *Theoretical Computer Science*, 382(2):139–150, 2007.

[62] Eugene L. Lawler, Jan K. Lenstra, and Alexander H. G. Rinnooy Kan. Generating all maximal independent sets: NP-hardness and polynomial-time algorithms. *SIAM Journal on Computing*, 9(3):558–565, 1980.

[63] John M. Lewis and Mihalis Yannakakis. The node-deletion problem for hereditary properties is NP-complete. *Journal of Computer and System Sciences*, 20(2):219–230, 1980.

[64] Arnaud Mary. *Énumération des dominants minimaux d'un graphe*. PhD thesis, Université Blaise Pascal, 2013. Courtesy of the author.

[65] Russell Merris. Split graphs. *European Journal of Combinatorics*, 24(4):413–430, 2003.

[66] John W. Moon and Leo Moser. On cliques in graphs. *Israel journal of Mathematics*, 3(1):23–28, 1965.

[67] Assaf Natanzon, Ron Shamir, and Roded Sharan. Complexity classification of some edge modification problems. *Discrete Applied Mathematics*, 113(1):109–128, 2001.

[68] Tatsuo Ohtsuki. A fast algorithm for finding an optimal ordering for vertex elimination on a graph. *SIAM Journal on Computing*, 5(1):133–145, 1976.

[69] Andreas Parra and Petra Scheffler. Characterizations and algorithmic applications of chordal graph embeddings. *Discrete Applied Mathematics*, 79(1-3):171–188, 1997.

[70] Seymour Parter. The use of linear graphs in gauss elimination. *SIAM review*, 3(2):119–130, 1961.

[71] Marvin C. Paull and Stephen H. Unger. Minimizing the number of states in incompletely specified sequential switching functions. *IRE Transactions on Electronic Computers*, (3):356–367, 1959.

[72] Ronald C. Read and Robert E. Tarjan. Bounds on backtrack algorithms for listing cycles, paths, and spanning trees. *Networks*, 5(3):237–252, 1975.

[73] Neil Robertson and P.D Seymour. Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984.

[74] Donald J. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In *Graph theory and computing*, pages 183–217. Elsevier, 1972.

[75] Donald J. Rose, R. Endre Tarjan, and George S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on computing*, 5(2):266–283, 1976.

[76] Frank Ruskey. *Combinatorial generation*. University of Victoria, Victoria, BC, Canada, 2003. https://page.math.tu-berlin.de/~felsner/SemWS17-18/Ruskey-Comb-Gen.pdf.

[77] Benno Schwikowski and Ewald Speckenmeyer. On enumerating all minimal solutions of feedback problems. *Discrete Applied Mathematics*, 117(1-3):253–265, 2002.

[78] George Stibitz. Binary counter, U. S. Patent 2 307 868, November 26, 1941.

[79] Yann Strozecki. Enumeration complexity and matroid decomposition, 2010. https://yann-strozecki.github.io/these_strozecki.pdf.

[80] Yann Strozecki et al. Enumeration complexity. *Bulletin of EATCS*, 3(129), 2019.

[81] David P. Sumner. Dacey graphs. *Journal of the Australian Mathematical Society*, 18(4):492–502, 1974.

[82] Robert E. Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13(3):566–579, 1984.

[83] Ioan Todinca. *Aspects algorithmiques des triangulations minimales des graphes*. PhD thesis, École normale supérieure (Lyon), 1999. https://www.univ-orleans.fr/lifo/Members/todinca/publis.html#others.

[84] Shuji Tsukiyama, Mikio Ide, Hiromu Ariyoshi, and Isao Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM Journal on Computing*, 6(3):505–517, 1977.

[85] Takeaki Uno. Two general methods to reduce delay and change of enumeration algorithms. Technical report, National Institute of Informatics, 2003. https://www.nii.ac.jp/TechReports/public_html/03-004E.pdf.

[86] Mihalis Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 2:77–79, 1981.

[87] Mihalis Yannakakis. Edge-deletion problems. *SIAM Journal on Computing*, 10(2):297–309, 1981.

Index

Summary

This thesis is about graph theory and more specifically enumeration algorithms for subgraphs or supergraphs. The problem we consider is the following: given a graph G and a graph property $\mathcal{P}$, can we find an efficient algorithm generating exactly once all the inclusion-wise maximal (induced or not) subgraphs of G that satisfy the property $\mathcal{P}$? We are interested as well in inclusion-wise minimal supergraphs of G – called minimal completions of G – verifying $\mathcal{P}$. Efficient algorithms, running with polynomial delay in polynomial space, are obtained for some of these problems when $\mathcal{P}$ describes the classes of split graphs, cographs, threshold graphs and chordal graphs.

The thesis is organised as follows. First, the principal existing enumeration techniques are presented. Then, we are interested in the split property, for which we give a polynomial delay polynomial space algorithm enumerating minimal completions and deletions (maximal non-induced subgraphs). We are then interested in the well known Flashlight Search technique and show that it cannot be used to enumerate efficiently the maximal subgraphs in any class. After that, the recent Proximity Search technique is applied to the enumeration of maximal induced sub-cographs, maximal induced threshold subgraphs and minimal deletions into threshold graphs. This way, we obtain polynomial delay exponential space algorithms. This technique is also successfully applied to the enumeration of minimal completions into chordal graphs, for which the existence of a polynomial delay algorithm was an open problem. Finally, a generalisation of Proximity Search is proposed, and a new technique based on Proximity Search for polynomial delay polynomial space algorithms is introduced. We apply this technique to most of the problems solved in this thesis *via* Proximity Search, therefore reducing their space complexity.

Résumé

Cette thèse porte sur la théorie des graphes et plus particulièrement les algorithmes d'énumération de sous-graphes ou sur-graphes. Le problème auquel on s'intéresse est le suivant : étant donné un graphe quelconque G et une propriété de graphes $\mathcal{P}$, peut-on trouver un algorithme efficace qui génère une et une seule fois tous les sous-graphes maximaux pour l'inclusion (induits ou non) de G qui vérifient la propriété $\mathcal{P}$? De même, on s'intéresse aussi aux sur-graphes de G minimaux pour l'inclusion – appelés complétions minimales de G – qui vérifient $\mathcal{P}$. Des algorithmes efficaces, à délai polynomial en espace polynomial, sont obtenus pour certains de ces problèmes lorsque $\mathcal{P}$ décrit la classe des graphes *split*, des cographes, des graphes *threshold* et des graphes cordaux.

La thèse se décompose de la manière suivante. D'abord, les principaux algorithmes d'énumération existants sont présentés. Puis on s'intéresse à la propriété *split*, pour laquelle on donne un algorithme à délai polynomial en espace polynomial pour l'énumération des complétions minimales et délétions minimales (sous-graphes maximaux non induits). On s'intéresse ensuite à une technique d'énumération existante appelée *Flashlight Search* et on montre qu'elle ne peut être utilisée pour énumérer efficacement les sous-graphes maximaux dans n'importe quelle classe. Après cela, la récente technique de *Proximity Search* est appliquée à l'énumération des sous-graphes maximaux induits dans la classe des cographes, des sous-graphes *threshold* maximaux induits et des délétions minimales en graphe *threshold*. Nous obtenons ainsi des algorithmes à délai polynomial en espace exponentiel. Cette technique est aussi appliquée avec succès à l'énumération des complétions minimales en graphe cordal, pour lesquelles l'existence d'un algorithme à délai polynomial était un problème ouvert. Enfin, une généralisation de la technique de *Proximity Search* est proposée, et une nouvelle technique fondée sur le *Proximity Search* pour des algorithmes à délai polynomial en espace polynomial est introduite. Nous appliquons cette technique à la plupart des problèmes résolus dans cette thèse *via Proximity Search*, ce qui réduit leur complexité en espace.