



Cliques statiques et temporelles : algorithmes d'énumération et de détection de communautés

Alexis Baudin

► To cite this version:

Alexis Baudin. Cliques statiques et temporelles : algorithmes d'énumération et de détection de communautés. Algorithme et structure de données [cs.DS]. Sorbonne Université, 2023. Français. NNT : 2023SORUS609 . tel-04511282

HAL Id: tel-04511282

<https://theses.hal.science/tel-04511282>

Submitted on 19 Mar 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Thèse présentée pour obtenir le grade de docteur

Sorbonne Université



École doctorale :

Informatique, Télécommunication et Électronique Paris (ED130)

Discipline : Informatique

**Cliques statiques et temporelles :
algorithmes d'énumération et de détection de
communautés**

PAR : Alexis Baudin

MEMBRES DU JURY:

Arnaud CASTEIGTS	Professeur des Universités, Université de Genève	<i>Rapporteur</i>
Vincent LABATUT	Maître de Conférences, LIA, Université d'Avignon	<i>Rapporteur</i>
Alix MUNIER	Professeure des Universités, LIP6, Sorbonne Université, CNRS	<i>Examinaterice</i>
Christian VESTERGAARD	Chargé de recherche, Institut Pasteur, CNRS	<i>Examinateur</i>
Clémence MAGNIEN	Directrice de recherche, LIP6, Sorbonne Université, CNRS	<i>Co-directrice</i>
Lionel TABOURIER	Maître de Conférences, LIP6, Sorbonne Université, CNRS	<i>Co-directeur</i>

Date de soutenance : 14 décembre 2023

Remerciements

Je tiens à remercier tout d'abord Clémence Magnien et Lionel Tabourier, qui ont encadré ma thèse. Merci pour votre implication et pour la façon dont vous avez su partager avec moi vos réflexions et vos connaissances scientifiques. Vous avez été très présents, en accompagnant patiemment mes travaux avec beaucoup de bienveillance et d'intérêt et j'ai énormément appris à vos côtés. En plus de la qualité de l'encadrement que vous m'avez fourni, nous avons passé de très bons moments, et cette bonne entente a fait de ces trois années un vrai plaisir à travailler ensemble. Vous avez été un soutien moral constant, et je ne peux que vous remercier d'avoir autant pris à cœur l'encadrement de ma thèse.

Je souhaite rendre hommage à Maximilien Danisch qui m'a accueilli en stage de Master 2, et a commencé à encadrer ma thèse, avant son décès brutal six mois plus tard. Sa présence quotidienne pendant cette année de travail en commun, durant laquelle nous avons partagé de très bons moments, a été à l'origine de mes connaissances, de mes intuitions et de ma motivation à travailler sur ces sujets de recherche, et je lui en suis profondément reconnaissant.

Je remercie les membres de mon jury, Arnaud Casteigts, Vincent Labatut, Alix Munier et Christian Vestergaard, qui ont accepté de lire et d'évaluer mon travail de thèse, et qui m'honorent de leur présence. Je remercie également Alix et Christian pour avoir pris le temps de participer à mes comités de suivi de thèse. Merci à Alix, avec qui j'ai eu l'opportunité d'enseigner en travaux dirigés pour la première fois, et avec qui j'ai beaucoup apprécié travailler. Merci aussi à Christian, pour les échanges interdisciplinaires avec son doctorant Alexis qui ont été importants pour moi à un moment où je cherchais à explorer davantage ce domaine.

Je remercie chaleureusement l'équipe Complex Networks, dans laquelle j'ai réalisé ma thèse. C'est une équipe où je me suis directement senti bien, et dans laquelle la bonne ambiance a permis de très bonnes conditions de travail. Un merci tout particulier à docteur Fabrice Lécuyer : chacune des étapes de la thèse, nous les avons vécues ensemble. Nous nous sommes accompagnés dans les bons moments et aussi dans nos difficultés, et sans cela je serais resté bloqué plus d'une fois. Merci à Matthieu Latapy pour sa bonne humeur permanente et sa vision engagée de la recherche qui m'inspire beaucoup. Merci à tous les doctorants et postdoctorants qui sont passés dans l'équipe à un moment de ma thèse. Je pense notamment à Hong-Lan et nos pauses goûter quand je suis arrivé dans l'équipe, puis Mehdi, Stephany, Esteban, Julien et Ioannis pour tous les moments de pauses et de jeu (pensées au Papayoo et au Star Realm qu'on a découvert et auxquels je joue encore !), et enfin les derniers doctorants Guillaume, Bastien, Imane, Duncan et Nouamane, merci pour

votre sympathie et je vous souhaite beaucoup de réussite dans vos recherches. Merci également à l'équipe des gestionnaires financiers, Aline, Sabine, Hélène, Laurence, Marie-Véronique et Stéphanie, très à l'écoute et réactives, qui m'ont aidé dans de nombreuses démarches administratives. Enfin, un grand merci à tout le personnel de nettoyage de l'université, sans qui nous ne pourrions pas travailler. Pensées affectives à leur combat pour des conditions de travail et une rémunération dignes.

Ma thèse n'aurait pas été la même sans les différentes collaborations scientifiques que j'ai eu la chance d'avoir. Merci à Élodie Laine, qui m'a fait découvrir et participer à son travail de recherche en bioinformatique avec beaucoup d'intérêt. Merci à Sergey Kirgizov, qui m'a accompagné avec Clémence Magnien dans la rédaction de mon premier article, dans un moment qui était difficile pour tout le monde ; une expérience qui m'a énormément appris et qui m'a permis d'aborder les travaux suivants avec plus de confiance (et ta compilation automatique des documents latex est maintenant indispensable pour moi!). Je remercie aussi Matthieu Latapy, Guillaume Moinard, Bastien Legay, et toute l'équipe du projet PRIMS pour leur accueil au séminaire de la Saline avec les collègues géographes.

Ces trois années ne se seraient pas si bien passées sans mes amis, qui ont été un pilier pour moi. Merci à Anatole, soutien quotidien, qui me partage sa passion de la magie depuis des années ; Pierre, Jérôme et Thomas pour les soirées jeux et pizzas tirées au dé, et le partage de nos doctorats respectifs ; et tous les copains avec qui j'ai mangé de très nombreuses soupes de nouilles à Trantranzai. Merci à Inès, Audrey et Arthur qui m'offrent une parenthèse de bonheur dès qu'on se voit. Merci à l'équipe TF2, et notamment Sirine, Julien, Manon, Tomek et Julian pour les tarots interminables, les concerts, et toutes les discussions et projets politiques qu'on a eus ensemble. Merci à tous les camarades avec qui j'ai partagé les travaux d'éducation populaire, et fait plusieurs événements et campagnes, ainsi que les camarades de Solidaires étudiant·es de Jussieu, qui luttent avec beaucoup d'énergie pour une université émancipatrice et solidaire. Enfin, je souhaite remercier Diane qui a supporté les confinements et couvre-feux en coloc avec moi pendant la première année de ma thèse, et avec qui je garde d'excellents souvenirs (et un traversin !).

Je tiens à remercier ma famille, en particulier mes parents et mes frère et sœurs Paul, Camille et Émilie, qui m'encouragent depuis le début de mes études, et représentent une base de repli précieuse. Un merci spécial à Paul, toujours là pour moi, qui me comprend et trouve les conseils justes pour me faire avancer, en plus de ses relectures lorsque j'ai besoin d'un regard extérieur sur mon travail. Merci à Christine et ta générosité pour le buffet de thèse.

Enfin, merci à Adèle, merci pour ton soutien et la joie que tu m'offres au quotidien. Pendant la thèse, tu m'as apporté beaucoup d'aide et de courage, jusqu'à m'écouter répéter sans fin des présentations et relire des morceaux détaillés de mon travail. Merci pour tous les moments merveilleux passés ensemble, et à venir.

Résumé

Les graphes sont des objets mathématiques qui permettent de modéliser des interactions ou connexions entre entités de types variés. Un graphe peut représenter par exemple un réseau social qui connecte les utilisateurs entre eux, un réseau de transport comme le métro où les stations sont connectées entre elles, ou encore un cerveau avec les milliards de neurones en interaction qu'il contient. Depuis quelques années, la forte dynamicité de ces structures a été mise en évidence, ainsi que l'importance de prendre en compte l'évolution temporelle de ces réseaux pour en comprendre le fonctionnement. Alors que de nombreux concepts et algorithmes ont été développés sur les graphes pour décrire des structures de réseaux statiques, il reste encore beaucoup à faire pour formaliser et développer des algorithmes pertinents pour décrire la dynamique des réseaux réels.

Ma thèse vise à mieux comprendre comment sont structurés les graphes massifs qui sont issus du monde réel et à développer des outils pour étendre notre compréhension à des structures évoluant dans le temps. Il a été montré que ces graphes ont des propriétés particulières, qui les distinguent des graphes théoriques ou tirés aléatoirement. Exploiter ces propriétés permet alors de concevoir des algorithmes pour résoudre certains problèmes difficiles beaucoup plus rapidement sur ces instances que dans le cas général.

La thèse se focalise sur les cliques, qui sont des groupes d'éléments tous connectés entre eux. Nous étudions l'énumération des cliques dans les graphes statiques et temporels et la détection de communautés qu'elles permettent de mettre en œuvre. Les communautés d'un graphe sont des ensembles de sommets tels qu'au sein d'une communauté, les sommets interagissent fortement entre eux, et peu avec le reste du graphe. Leur étude aide à comprendre les propriétés structurelles et fonctionnelles des réseaux. Nous évaluons nos algorithmes sur des graphes massifs issus du monde réel, ouvrant ainsi de nouvelles perspectives pour comprendre les interactions au sein de ces réseaux.

Nous travaillons d'abord sur des graphes, sans tenir compte de la composante temporelle des interactions. Nous commençons par utiliser la méthode de détection de communautés par percolation de cliques, en mettant en évidence ses limites en mémoire, qui empêchent de l'appliquer à des graphes trop massifs. En introduisant un algorithme de résolution approchée du problème, nous dépassons cette limite. Puis, nous améliorons l'énumération des cliques maximales dans le cas des graphes particuliers dits bipartis. Ils correspondent à des interactions entre des groupes de

sommets de type différent, par exemple des liens entre des personnes et du contenu consulté, la participation à des événements, etc.

Ensuite, nous considérons des interactions qui ont lieu au cours du temps, grâce au formalisme des flots de liens. Nous cherchons à étendre les algorithmes présentés en première partie, pour exploiter leurs avantages dans l'étude des interactions temporelles. Nous fournissons un nouvel algorithme d'énumération des cliques maximales dans les flots de liens, beaucoup plus efficace que l'état de l'art sur des jeux de données massifs. Enfin, nous nous intéressons aux communautés dans les flots de liens par percolation de cliques, en développant une extension de la méthode utilisée sur les graphes. Les résultats montrent une amélioration significative par rapport à l'état de l'art, et nous analysons les communautés obtenues pour fournir des informations pertinentes sur l'organisation des interactions temporelles dans les flots de liens.

Mon travail de thèse a permis d'apporter de nouvelles réflexions sur l'étude des réseaux massifs issus du monde réel. Cela montre l'importance d'explorer le potentiel des graphes dans un contexte réel, et pourrait contribuer à l'émergence de solutions novatrices pour les défis complexes de notre société moderne.

Abstract

Graphs are mathematical objects used to model interactions or connections between entities of various types. A graph can represent, for example, a social network that connects users to each other, a transport network like the metro where stations are connected to each other, or a brain with the billions of interacting neurons it contains. In recent years, the dynamic nature of these structures has been highlighted, as well as the importance of taking into account the temporal evolution of these networks to understand their functioning. While many concepts and algorithms have been developed on graphs to describe static network structures, much remains to be done to formalize and develop relevant algorithms to describe the dynamics of real networks.

My thesis aims to better understand how massive graphs are structured in the real world, and to develop tools to extend our understanding to structures that evolve over time. It has been shown that these graphs have particular properties, which distinguish them from theoretical or randomly drawn graphs. Exploiting these properties then enables the design of algorithms to solve certain difficult problems much more quickly on these instances than in the general case.

My PhD thesis focuses on cliques, which are groups of elements that are all connected to each other. We study the enumeration of cliques in static and temporal graphs and the detection of communities they enable. The communities of a graph are sets of vertices such that, within a community, the vertices interact strongly with each other, and little with the rest of the graph. Their study helps to understand the structural and functional properties of networks. We are evaluating our algorithms on massive real-world graphs, opening up new perspectives for understanding interactions within these networks.

We first work on graphs, without taking into account the temporal component of interactions. We begin by using the clique percolation method of community detection, highlighting its limitations in memory, which prevent it from being applied to graphs that are too massive. By introducing an approximate problem-solving algorithm, we overcome this limitation. Next, we improve the enumeration of maximal cliques in the case of bipartite graphs. These correspond to interactions between groups of vertices of different types, e.g. links between people and viewed content, participation in events, etc.

Next, we consider interactions that take place over time, using the link stream

formalism. We seek to extend the algorithms presented in the first part, to exploit their advantages in the study of temporal interactions. We provide a new algorithm for enumerating maximal cliques in link streams, which is much more efficient than the state-of-the-art on massive datasets. Finally, we focus on communities in link streams by clique percolation, developing an extension of the method used on graphs. The results show a significant improvement over the state of the art, and we analyze the communities obtained to provide relevant information on the organization of temporal interactions in link streams.

My PhD work has provided new insights into the study of massive real-world networks. This shows the importance of exploring the potential of graphs in a real-world context, and could contribute to the emergence of innovative solutions for the complex challenges of our modern society.

Table des matières

Remerciements	3
Résumé	5
Abstract	7
1 Introduction	13
2 Préliminaires et état de l'art	21
2.1 Définitions de base	21
2.1.1 Graphes	21
2.1.2 Graphes bipartis	23
2.1.3 Flots de liens	23
2.2 Les graphes issus du monde réel	24
2.2.1 Propriétés des graphes du monde réel	25
2.2.2 Expression des complexités avec des variables pertinentes	26
2.2.3 Valider les algorithmes par une étude expérimentale	28
2.3 Modéliser les réseaux temporels	28
2.4 Description de la structure locale des graphes	32
2.4.1 Motifs	32
2.4.2 Sous-graphes denses, k -cliques, k -plexes	33
2.4.3 Détection de communautés	35
3 Communautés par percolation de cliques	39
3.1 Introduction	39
3.2 Définition de CPM et algorithme	41
3.2.1 Structure de données Union-Find	42
3.2.2 Algorithme CPM	46
3.2.3 Complexité de l'algorithme CPM	49
3.3 État de l'art	49
3.4 Représenter les communautés par leurs z -cliques	50
3.4.1 Stocker les z -cliques plutôt que les $(k - 1)$ -cliques	51
3.4.2 $(k - 1)$ -cliques parasites d'un ensemble de z -cliques	51
3.4.3 Définition des communautés CPMZ	52
3.5 Union-Find sur des ensembles non disjoints	54
3.6 Algorithme pour calculer les communautés CPMZ	57

3.7	Analyse	59
3.7.1	Validité de l'algorithme CPMZ	59
3.7.2	Complexité de l'algorithme CPMZ	60
3.8	Évaluation expérimentale	61
3.8.1	Comparaison de notre implémentation CPM à l'état de l'art	62
3.8.2	Étude du gain en mémoire et du coût en temps de l'algorithme CPMZ	65
3.8.3	Similarités entre communautés CPM et CPMZ	67
3.9	Conclusion et discussion	68
4	Énumération des bicliques maximales	71
4.1	Introduction	71
4.2	Énumération des cliques maximales dans les graphes	73
4.2.1	Algorithmes de Bron-Kerbosch et Eppstein <i>et al.</i>	74
4.2.2	Complexité en fonction de la sortie	77
4.2.3	Travaux récents sur l'énumération des cliques maximales	80
4.3	État de l'art : énumération des bicliques maximales	81
4.4	Algorithme d'énumération des bicliques maximales	82
4.4.1	Graphe étendu d'un graphe biparti	83
4.4.2	BBK : nouvel algorithme d'énumération des bicliques maximales	85
4.4.3	Complexité de l'algorithme	88
4.5	Évaluation expérimentale	89
4.5.1	Jeux de données	90
4.5.2	Résultats : temps de calcul	92
4.5.3	Énumérer les sommets à partir de U ou de V ?	96
4.5.4	Impact de l'ordre sur le temps de l'énumération	98
4.5.5	Discussion sur la mémoire	99
4.6	Conclusion	101
5	Cliques maximales dans les flots de liens	103
5.1	Introduction	104
5.2	Définitions et notations	105
5.2.1	Rappels sur les flots de liens	105
5.2.2	Graphe instantané d'un flot de liens au temps t	105
5.3	État de l'art	107
5.4	Algorithme d'énumération des cliques maximales	108
5.4.1	Structure générale de l'algorithme	109
5.4.2	Énumération des cliques dans les graphes instantanés G_t	112
5.4.3	Filtrer les cliques maximales en sommets parmi les cliques maximales en temps	113
5.4.4	Pivot pour améliorer l'énumération des cliques dans les graphes G_t	115
5.5	Analyse de l'algorithme	117

5.5.1	Validité	118
5.5.2	Complexité en fonction des caractéristiques de l'entrée et de la sortie de l'algorithme	121
5.6	Évaluation expérimentale	129
5.6.1	Dispositif expérimental	129
5.6.2	Comparaison à l'état de l'art	130
5.6.3	Passage à l'échelle sur des flots de liens massifs issus du monde réel	135
5.6.4	Mesure de l'efficacité du pivot	137
5.6.5	Expérimentations avec la version parallèle de l'algorithme	141
5.7	Conclusion	143
6	LSCPM : percolation de cliques temporelles	145
6.1	Introduction	146
6.2	Définitions et notations	147
6.2.1	Cliques dans les flots de liens	147
6.2.2	Communautés dans les flots de liens	148
6.3	Énumération des k -cliques dans les flots de liens	150
6.4	Union-Find sur des ensembles d'éléments temporels	154
6.5	LSCPM : algorithme CPM adapté aux flots de liens	156
6.5.1	Pseudocode	157
6.5.2	Complexité	159
6.6	Étude expérimentale	160
6.6.1	Jeux de données	161
6.6.2	Performances de LSCPM et passage à l'échelle de données massives	163
6.6.3	Comparaison entre les communautés LSCPM et DCPM	166
6.6.4	Informations fournies par les communautés LSCPM	168
6.6.5	Influence de k sur la structure de communautés	172
6.6.6	Mesurer le taux de dynamité d'une communauté	175
6.7	Conclusion et perspectives	176
7	Conclusion et perspectives	179
7.1	Contributions	179
7.2	Perspectives	181
A	Collaboration interdisciplinaire	187
Bibliographie		201

Introduction

Les réseaux sont omniprésents dans notre environnement quotidien et notre société moderne repose sur des structures interconnectées. Par exemple, notre vie collective dépend des réseaux d'approvisionnement en eau potable, de l'accès à l'électricité à domicile et à Internet, ainsi que des réseaux de transport pour notre mobilité ou les transactions commerciales. Qu'il s'agisse de nos interactions physiques ou sur les réseaux sociaux, nous sommes en interaction quasi constante les uns avec les autres. Les réseaux ont aussi une importance significative dans des domaines comme la biologie, où ils jouent un rôle clé dans la propagation des maladies, l'activité neuronale d'un cerveau ou les interactions entre protéines, pour n'en citer que quelques-uns. La liste des exemples pourrait encore être longue. Le point commun à tous ces cas est l'interaction entre des éléments organisés en réseaux, sans nécessité d'un groupe de contrôle central. C'est ce concept qui est représenté par la notion de graphe.

Un graphe est un objet mathématique qui permet de modéliser des connexions entre entités de types variés. Il se compose de deux éléments fondamentaux : les *nœuds* (ou *sommets*) qui représentent les entités, et les *arêtes* qui expriment les connexions entre ces entités. Les graphes offrent un formalisme mathématique puissant pour décrire les structures et les propriétés de ces connexions, ouvrant la voie à des analyses, des modélisations et des résolutions de problèmes variés.

Les graphes issus du monde réel revêtent une importance capitale pour comprendre la complexité des systèmes qui nous entourent. Par exemple, dans les domaines des réseaux de communication ou de transport, une compréhension de la topologie des graphes permet de concevoir des infrastructures résilientes face aux pannes et aux perturbations. Dans les sciences sociales, l'analyse des réseaux sociaux aide à comprendre les interactions entre les individus, les groupes et les organisations. En biologie, les réseaux d'interactions entre les protéines, les gènes ou les molécules sont essentiels pour comprendre les processus biologiques et les maladies. L'étude de ces graphes nous permet donc de mieux appréhender leur organisation, leur évolution, leurs propriétés émergentes et leur comportement global.

L'une des observations principales sur lesquelles se base cette thèse est que les graphes rencontrés dans des situations réelles ne sont pas arbitraires. Des études ont montré que ces graphes présentent des caractéristiques qui les différencient des graphes tirés aléatoirement [BA99 ; New03]. Par exemple, ils sont généralement peu denses dans l'ensemble, mais peuvent être densément connectés localement. Cette densité locale se manifeste, par exemple, dans les réseaux sociaux par la formation de groupes d'amis. Il a également été constaté que la distribution des connexions des nœuds est hétérogène, avec la présence de nœuds très connectés, appelés *hubs*, ainsi que de nombreux nœuds peu connectés. Exploiter ces propriétés spécifiques dans les algorithmes permet d'obtenir de bonnes performances pour des problèmes complexes en théorie, qui ne pourraient pas être résolus sur des grands graphes dans le cas général. Il a ainsi été observé que de nombreux algorithmes présentant une complexité en temps élevée dans le pire des cas s'exécutent en fait de manière rapide sur les graphes du monde réel. Par exemple, le caractère peu dense d'un réseau social permet de faire des opérations efficaces sur les ensembles d'amis, qui sont de petite taille par rapport à la taille globale du réseau. Un problème classique consiste à énumérer des amis des membres du réseau qui se connaissent entre eux (énumérer les triangles).

Dans de nombreux réseaux réels, les interactions ne restent pas figées, mais évoluent avec le temps. Ces réseaux présentent une structure dynamique, où les connexions entre les entités sont soumises à des variations temporelles. Par exemple, si on considère le réseau social *Instagram*, les liens entre les utilisateurs se créent au fil des rencontres ou des découvertes d'intérêts partagés, et parfois disparaissent, ce qui n'est pas capturé par une simple représentation statique du réseau. À deux instants différents, le graphe associé à ce réseau social existe, mais n'est pas le même. D'autre part, d'autres réseaux présentent des interactions au fil du temps sans avoir de graphe évident à analyser à un instant donné, comme c'est le cas des envois de mails entre des utilisateurs. Ces deux types de connexions temporelles peuvent influer les unes sur les autres. Si on considère un réseau social, les liens d'amitiés entre les personnes forment un réseau dont la structure évolue avec le temps, et les interactions sociales, plus dynamiques, ont un impact sur ces liens d'amitiés. Dans un autre contexte, dans un réseau de transport, il existe généralement un réseau d'itinéraires et un ensemble d'unités de transport qui circulent sur ces itinéraires au fil du temps. Cette perspective temporelle est également essentielle pour comprendre la propagation d'épidémies, où les contacts entre individus et la diffusion d'une maladie infectieuse sont intrinsèquement liés au temps.

L'analyse des réseaux d'interaction du monde réel a progressé de manière significative ces dernières années par le développement des approches dynamiques, ouvrant la voie aux graphes temporels, et au développement d'outils d'analyse adaptés. Cette évolution s'est appuyée sur l'accessibilité accrue aux données temporelles pour

lesquelles chaque interaction est horodatée. Dans cette thèse, nous utilisons une modélisation des réseaux temporels sous la forme de *flots de liens*, qui représentent les liens entre des paires de sommets, chacun associé à un intervalle de temps d'existence. Ainsi, notre travail concerne à la fois les graphes statiques et les réseaux temporels sous forme de flots de liens.

Grâce aux avancées technologiques récentes, nous avons accès à une quantité croissante de données. L'explosion en taille et en nombre de données disponibles entraîne le besoin de développer des algorithmes rapides pour les analyser. En effet, nous sommes désormais fréquemment confrontés à des graphes, temporels ou non, contenant des millions, voire des milliards d'interactions. Par exemple, *Google* indexe près de 130 milliards de pages web, ou encore un cerveau humain contient plus de 100 milliards de connexions neuronales. Le traitement de ces données par des algorithmes pose des défis, tant en termes de stockage en mémoire que de temps de calcul. Ainsi, de nombreux algorithmes existants ne sont pas directement applicables à l'échelle de ces réseaux massifs. Cela nous conduit à explorer la nécessité d'adapter les méthodes d'étude des graphes, de manière à ce qu'elles puissent traiter efficacement ces volumes considérables d'interactions.

Étant donnée la nature complexe de ces réseaux contenant beaucoup d'interactions, il n'est pas évident de les décrire de manière concise et intuitive en vue d'une analyse efficace. Un problème très étudié pour comprendre les interactions dans les réseaux massifs est celui de l'organisation de ses sommets. Il est intéressant de regrouper les sommets en ensembles pertinents, afin d'une part d'en faire une synthèse qui permet une compréhension globale du réseau, et d'autre part de cibler des sous-ensembles d'intérêt à analyser. Dans cette optique, la détection de communautés est un problème fondamental. Elle consiste à identifier des groupes de noeuds densément connectés, mais faiblement connectés au reste du graphe. De manière plus générale, les communautés peuvent évoluer dans le temps et leur étude est également fondamentale dans les réseaux temporels. Dans la littérature, la détection de communautés prend différentes formes : on peut chercher à obtenir une partition où chaque noeud appartient à une seule communauté ou opter pour une structure recouvrante où un noeud peut appartenir simultanément à plusieurs communautés. Il est également possible de rechercher les communautés spécifiques auxquelles appartient un noeud particulier, plutôt que de chercher à décrire l'ensemble du réseau. Dans les réseaux temporels, ce sont les sommets temporels qui sont organisés en communautés : un sommet peut appartenir à une communauté sur une période donnée.

La détection de communautés trouve des applications directes dans divers domaines. Par exemple, dans les réseaux sociaux tels qu'*Instagram* ou *Twitter*, elle aide à repérer des groupes de personnes partageant des intérêts similaires. Cela peut avoir des implications pour la personnalisation du contenu, ou pour la publicité

ciblée par exemple. Plus précisément, dans le domaine du commerce en ligne, la recommandation de produits similaires à ceux déjà consultés ou achetés peut reposer sur l'identification de groupes de clients aux préférences semblables. Dans le domaine de la biologie, il est possible par exemple de prédire les fonctions d'une protéine inconnue en examinant les communautés de protéines connues avec lesquelles elle interagit : si elle apparaît dans la communauté de protéines connues, alors ceci donne des éléments pour déterminer sa fonction. Un autre domaine d'application majeur est la classification de documents. En examinant les articles de *Wikipédia*, les publications scientifiques ou même les textes de lois, on constate souvent que ces documents contiennent des références croisées et des connexions entre eux. En utilisant des techniques de détection de communautés, il devient possible de regrouper ces documents en fonction de leurs similarités thématiques, ce qui facilite la recherche d'informations et la navigation.

Une approche pertinente pour identifier les communautés consiste à rechercher des motifs denses dans le réseau. Le sous-graphe le plus dense qui existe est la *clique*, dont tous les sommets sont connectés entre eux, et qui peut être utilisé comme brique de base pour la recherche de communautés. En effet, en définissant une règle d'adjacence des cliques lorsqu'elles partagent suffisamment de sommets, cela permet de partitionner l'ensemble des cliques de manière adéquate pour former les communautés. Cette méthode est appelée la détection de communautés par *percolation de cliques*. De plus, il est important de noter que les cliques peuvent évoluer dans le temps au sein d'un réseau. Par exemple, un groupe d'amis peut s'agrandir avec de nouvelles rencontres, ou au contraire se réduire par des accidents ou mésaventures. Nous verrons ainsi que les cliques nous permettent d'obtenir des communautés, statiques et temporelles.

Cette thèse se concentre spécifiquement sur les cliques, qui sont des structures importantes pour la compréhension des graphes du monde réel. Notre recherche porte sur l'énumération des cliques dans les graphes, statiques et temporels, ainsi que sur la détection des communautés qu'on peut en tirer. Les problèmes d'énumération s'intéressent aux cliques maximales, c'est-à-dire qui ne sont incluses dans aucune autre, pour éviter les redondances dans l'énumération (toute clique est incluse dans une clique maximale). L'énumération des cliques de taille fixée est également un problème d'intérêt, notamment pour les grands graphes, car dans ce contexte, il est en général plus rapide à résoudre que celui de l'énumération de cliques maximales, qui est exponentiel. L'algorithme le plus célèbre pour énumérer les cliques maximales dans les graphes est celui de Bron-Kerbosch. Il est reconnu comme la base des méthodes les plus efficaces en pratique, et on s'en sert dans cette thèse comme point de départ à nos travaux d'énumération. Des méthodes ont également été établies pour énumérer les cliques maximales dans les réseaux temporels. Dans cette thèse, nous utilisons les cliques comme des éléments de base pour construire des commu-

nautés, en mettant en œuvre une méthode de percolation de cliques. L'efficacité de nos algorithmes constitue le cœur de nos recherches.

Dans la pratique, nous parvenons à traiter efficacement des graphes massifs, générant des résultats rapidement grâce à nos algorithmes, parfois même dans des cas où aucune autre méthode existante n'est en mesure de le faire. Nous expérimentons nos algorithmes en utilisant des graphes massifs issus du monde réel. Ces algorithmes ont pour finalité de montrer comment les cliques peuvent aider à comprendre l'organisation des interactions au sein des réseaux issus du monde réel.

Plan de la thèse :

Le chapitre 2 présente les définitions préliminaires sur les graphes et le formalisme des flots de liens que nous utilisons pour modéliser les graphes temporels. Ensuite, il présente l'état de l'art autour des graphes issus du monde réel en présentant leurs propriétés particulières qui peuvent être exploitées pour améliorer la complexité en pratique et les performances des algorithmes ; puis il introduit l'état de l'art de l'étude des réseaux temporels. Enfin, nous présentons l'état de l'art sur les méthodes de description des réseaux issus de données réelles, à travers la recherche de motifs, de sous-graphes denses, puis l'étude des cliques et des communautés.

Le chapitre 3 porte sur la détection de communautés dans les graphes, par percolation de cliques. Cette méthode calcule des communautés qui se chevauchent, en se basant sur un partitionnement des k -cliques du graphe. En utilisant l'un des meilleurs algorithmes d'énumération de k -cliques dans les graphes, nous en fournissons une nouvelle implémentation, plus efficace que celles de l'état de l'art. La limite à ce calcul de communautés est son besoin en mémoire, qui l'empêche de passer à l'échelle de graphes massifs issus du monde réel. Nous introduisons alors un algorithme capable de calculer des communautés très proches de celles-ci et qui est moins coûteux en mémoire. Nous en faisons une analyse théorique, puis nous présentons des expériences, afin d'illustrer le fait que les communautés obtenues sont proches de celles de l'état de l'art, et de comparer les performances des différentes méthodes. Nous montrons en particulier que le nouvel algorithme que nous développons permet d'obtenir des communautés sur des jeux de données pour lesquels aucun algorithme de l'état de l'art ne le permet.

Le chapitre 4 propose un nouvel algorithme d'énumération des bicliques maximales dans le cas des graphes bipartis. Cet algorithme est plus simple que ceux proposés par l'état de l'art, et il permet de réaliser l'énumération sur des graphes massifs impossibles à traiter avec les implementations existantes. Dans nos expériences, nous montrons le fait que l'ordre des sommets et le choix du côté du graphe biparti sur lequel l'énumération est lancée ont un impact

sur le temps de calcul. En particulier, nous constatons que le comportement de l'algorithme n'est pas le même sur les graphes les plus massifs que sur les petits graphes de nos jeux de données, ce qui est intéressant pour l'étude des graphes bipartis du monde réel.

Le chapitre 5 développe un nouvel algorithme d'énumération des cliques maximales dans les flots de liens. Pour résoudre ce problème efficacement, nous faisons correspondre les cliques du flot de liens avec les cliques des graphes des interactions instantanées. Cela permet de réduire considérablement les opérations par rapport aux travaux existants, favorisant ainsi le calcul sur des flots de liens massifs. Nous faisons une étude théorique complète de l'algorithme, en montrant sa validité ainsi que deux formules de complexité : une en fonction de l'entrée de l'algorithme et une en fonction de la sortie. Nous confirmons son efficacité par une étude expérimentale, qui montre un gain de temps de calcul de plusieurs ordres de grandeur par rapport à l'état de l'art.

Le chapitre 6 donne une nouvelle définition de communautés dans les flots de liens. Nous définissons une extension des k -cliques pour ce formalisme, et nous développons un algorithme pour les énumérer. Nous présentons ensuite un algorithme pour calculer ces communautés temporelles. Pour cela, nous partons de la méthode de percolation de cliques (CPM) présentée dans le chapitre 3, en présentant une méthode pour l'adapter au calcul de communautés temporelles. Nous comparons cet algorithme à l'extension de la percolation de cliques qui a été proposée dans les graphes temporels vus comme des suites de graphes instantanés. Notre nouvelle définition donne des communautés proches de celles de l'état de l'art et le temps de calcul pour les obtenir est plus rapide de plusieurs ordres de grandeur. Nous validons la pertinence de ces communautés temporelles dans des scénarios réels et mettons en évidence leur capacité à fournir des informations sur l'importance des noeuds dans les flots de liens.

Le chapitre 7 conclut la thèse et en présente les perspectives.

Publications :

Chapitre 3 [Bau+22] : *Clique percolation method : memory efficient almost exact communities*, Alexis Baudin, Maximilien Danisch, Sergey Kirgizov, Clémence Magnien et Marwan Ghanem.

Actes de la conférence internationale *Advanced Data Mining and Applications : 17th International Conference, ADMA 2021*, en visioconférence à Sydney le 2 février 2022 ;

Chapitre 5 [BMT23a] : *Énumération efficace des cliques maximales dans les flots de liens réels massifs*, Alexis Baudin, Clémence Magnien, et Lionel Tabourier.

Actes de la conférence nationale *23ème conférence francophone sur l'extraction et la gestion des connaissances, EGC 2023*, à Lyon le 16 janvier 2023 ;

Chapitre 6 [BTM23] : *LSCPM : communities in massive real-world Link Streams by Clique Percolation Method*, Alexis Baudin, Lionel Tabourier et Clémence Magnien.

Actes de la conférence internationale *30th International Symposium on Temporal Representation and Reasoning, TIME 2023*, à Athènes, le 25 septembre 2023 ;

Annexe A [Zea+21] : *Assessing conservation of alternative splicing with evolutionary splicing graphs*, Diego Javier Zea, Sofya Laskina, Alexis Baudin, Hugues Richard, et Élodie Laine.

Journal international *Genome Research, 2021*.

Article soumis :

Chapitre 5 [BMT23b], version longue de [BMT23a] : *Faster maximal clique enumeration in large real-world link streams*, Alexis Baudin, Clémence Magnien et Lionel Tabourier.

Soumis au journal international *Journal of Graph Algorithms and Applications*

Présentations invitées :

Chapitre 3 : Laboratoire d'Informatique de Bourgogne (LIB), à Dijon, le 16 décembre 2021 ;

Chapitre 6 : Institut du Cerveau (ICM), à Paris, le 21 juin 2023.

Codes en libre accès :

Chapitre 3 : <https://gitlab.lip6.fr/baudin/cpm-cpmz>

Chapitre 4 : <https://gitlab.lip6.fr/baudin/bbk>

Chapitre 5 : <https://gitlab.lip6.fr/baudin/maxcliques-linkstream>

Chapitre 6 : <https://gitlab.lip6.fr/baudin/lscpm>

Préliminaires et état de l'art

Dans ce chapitre, nous présentons le contexte du travail réalisé dans cette thèse. Dans la section 2.1, nous donnons les définitions de base que nous utilisons. Dans la section 2.2, nous présentons le cas spécifique des graphes issus du monde réel, et pourquoi leur étude se différencie de celle des graphes dans le cas général. Ensuite, dans la section 2.3, nous présentons l'état de l'art sur la notion de graphe temporel, que nous étudions dans la thèse avec le formalisme des flots de liens. Enfin, dans la section 2.4, nous présentons l'état de l'art des problèmes d'énumération de motifs, de sous-graphes denses et de communautés dans les graphes statiques ou temporels.

2.1 Définitions de base

2.1.1 Graphes

Un *graphe* est une paire $G = (V, E)$, où V est un ensemble de *sommets* et E un ensemble d'*arêtes* qui relient ces sommets entre eux. Une arête est de la forme $\{u, v\}$, où u et v sont des sommets distincts de V . Les graphes que nous considérons dans cette thèse sont des graphes simples, pour lesquels il n'y a pas plus d'une arête par paire de sommet. Ils sont non orientés : l'arête $\{u, v\}$ est la même que l'arête $\{v, u\}$, et sans boucle, c'est-à-dire qu'il n'y a pas d'arête reliant un sommet à lui-même. Dans la suite de cette section, on fixe $G = (V, E)$ un tel graphe. On note n le nombre de sommets de G ($n = |V|$), et m son nombre d'arêtes ($m = |E|$).

Soit $u \in V$ un sommet de G . On note $N(u)$ l'ensemble des sommets adjacents à u dans G . C'est ce que l'on appelle le *voisinage* de u dans G : $N(u) = \{v \in V \mid \{u, v\} \in E\}$. Notez que comme G est non orienté, on sait que si $v \in N(u)$, alors $u \in N(v)$. On dit alors que u est *voisin* de v , ou de manière symétrique que v est voisin de u , ou encore que u et v sont voisins. On appelle le *degré* de u , noté $d(u)$, le nombre de voisins de u dans G : $d(u) = |N(u)|$. Le *degré maximal* de G , que l'on note d , est alors le degré maximal qu'un de ses sommets atteint : $d = \max_{u \in V}(d(u))$.

Les deux manières les plus courantes pour stocker un graphe sont les *listes d'adjacences* et la *matrice d'adjacence*. La structure de listes d'adjacence associe à chaque sommet la liste de ses voisins. La matrice d'adjacence est une matrice $M = (M_{ij})$

à n lignes et n colonnes telle qu'en indiquant les sommets $V = \{v_1, \dots, v_n\}$, M_{ij} vaut 1 lorsqu'il y a une arête entre le sommet v_i et le sommet v_j , et vaut 0 sinon. Cette matrice est symétrique (puisque le graphe est non orienté).

Un *chemin* entre deux sommets u et v est une suite d'arêtes du graphe consécutives qui relient u à v : $((u_1, v_1), \dots, (u_p, v_p)) \in E^p$ avec $\forall i \in \{1, \dots, p-1\}$, $v_i = u_{i+1}$, et $u_1 = u$ et $v_p = v$. La *longueur* du chemin est égale au nombre d'arêtes p qui le composent. La *distance* entre u et v correspond à la longueur minimale d'un chemin entre u et v . On dit que G est *connexe* lorsqu'il existe un chemin entre chacune de ses paires de sommets. Une *composante connexe* de G est un sous-graphe induit connexe, qui est maximal, c'est-à-dire qu'il n'est inclus dans aucun autre sous-graphe induit connexe.

La *densité* du graphe est une valeur entre 0 et 1, qui correspond à la proportion d'arêtes qu'il contient par rapport au nombre maximal qu'il pourrait en contenir si tous les sommets étaient directement interconnectés. Un graphe contient au plus $\frac{n(n-1)}{2}$ arêtes, donc la densité d'un graphe vaut $\frac{2m}{n(n-1)}$. Plus la densité est proche de 1, et plus le graphe est dit *dense*. Au contraire, plus elle est faible, et plus le graphe est *clairsemé* (ou *creux*).

On dit que $H = (V', E')$ est un *sous-graphe* de G lorsque $V' \subseteq V$ et $E' \subseteq E$. H est un sous-graphe *induit* lorsque E' contient toutes les arêtes de G dont les sommets sont des sommets de H , *i.e.* lorsque $E' = \{\{u, v\} \in E \mid u, v \in V'\}$. On dit alors que H est le sous-graphe de G induit par V' . Un *sous-graphe dense* de G est un sous-graphe dont la densité est significativement plus élevée que celle de G . La densité des sous-graphes peut être reliée à la notion de k -core d'un graphe. Le k -core de G est le sous-graphe induit $H = (V', E')$ avec V' de taille maximale et tel que chaque sommet $u \in V'$ a un degré d'au moins k dans H . On appelle alors la *dégénérescence* d'un sommet $u \in V$, ou *core value* de u , la plus grande valeur de k telle que u appartienne au k -core de G . On note cette valeur $c(u)$. La *dégénérescence* ou *core value* de G est alors le maximum des core values de ses sommets, et on la note c . La core value a été introduite pour la première fois en 1983 par Seidman *et al.* [Sei83], et elle est très utilisée pour mesurer le caractère imbriqué de la structure du réseau. Nous revenons dessus dans la section 2.2, ainsi que dans le chapitre 4.

Enfin, nous introduisons la notion de *clique*, qui est la structure de base sur laquelle s'appuie l'ensemble de cette thèse. Une clique C d'un graphe $G = (V, E)$ est un ensemble de sommets de V qui sont tous connectés entre eux : $\forall u, v \in C$ ($u \neq v$), $\{u, v\} \in E$. Une k -*clique* est une clique contenant k sommets, *i.e.* $|C| = k$. On appelle *voisinage d'une clique* C , que l'on note \mathcal{N}_C , l'ensemble des sommets adjacents à tous les sommets de cette clique : $\mathcal{N}_C = \bigcap_{u \in C} N(u)$. On dit que C est *maximale* lorsqu'elle n'est incluse dans aucune autre clique de G , c'est-à-dire lorsqu'il n'existe aucune clique $C' \neq C$ telle que $C \subset C'$. Autrement dit, une clique est *maximale* si son voisinage est vide. Notez que les sommets d'une k -clique appartiennent néces-

sairement au $(k - 1)$ -core du graphe.

2.1.2 Graphes bipartis

Un *graphe biparti* est un triplet $G = (U, V, E)$, où U et V sont deux ensembles de sommets disjoints, et E est un ensemble d'arêtes entre des éléments de U et des éléments de V : $E \subseteq U \times V$. Le modèle des graphes bipartis est utilisé pour modéliser des interactions entre des entités de types différents, comme le lien entre des articles de recherche et leurs auteurs, les utilisateurs d'une plateforme de vidéo et les vidéos qu'ils consultent, la participation à des événements, etc. [GL06].

Un graphe biparti est en particulier un graphe $G' = (U \cup V, E)$, pour lequel les notions de la section précédente 2.1.1 sont les mêmes, à l'exception de la notion de clique. En effet, les sommets de U n'étant pas connectés entre eux, ni les sommets de V entre eux, cette notion doit être adaptée pour être pertinente dans ce cas. On appelle alors une *biclique* une paire (C_U, C_V) où C_U est un ensemble de sommets de U , C_V un ensemble de sommets de V , tels que tous les sommets de C_U sont connectés à tous les sommets de C_V : $\forall u \in C_U, \forall v \in C_V, (u, v) \in E$. Une (k_U, k_V) -*biclique* est telle que $|C_U| = k_U$ et $|C_V| = k_V$. De la même manière que dans les graphes, une biclique (C_U, C_V) est dite *maximale* lorsqu'elle n'est incluse dans aucune autre biclique : il n'existe aucune biclique $(C'_U, C'_V) \neq (C_U, C_V)$ telle que $C_U \subseteq C'_U$ et $C_V \subseteq C'_V$. Néanmoins, notez qu'il peut tout de même exister une biclique contenant C_U , ou contenant C_V .

Le *projeté* d'un graphe biparti $G = (U, V, E)$ sur U est le graphe $G_2(U) = (U, E_2(U))$, tel que deux sommets u_1 et u_2 forment une arête dans $E_2(U)$ lorsqu'ils ont un voisin en commun $v \in V$ dans G : $E_2(U) = \{\{u_1, u_2\} \mid u_1, u_2 \in U, u_1 \neq u_2 \text{ et } N(u_1) \cap N(u_2) \neq \emptyset\}$. $G_2(V)$ est défini de manière symétrique sur les sommets de V .

Nous appelons le *voisinage projeté* d'un $x \in U \cup V$, que nous notons $N_2(x)$, l'ensemble des sommets voisins d'au moins un voisin de x . Notez que cet ensemble contient x . Ainsi, le voisinage d'un sommet u dans $G_2(U)$ est $N_2(u) \setminus \{u\}$.

2.1.3 Flots de liens

Pour modéliser les interactions ayant lieu sur des intervalles de temps, nous utilisons le formalisme des flots de liens [LVM18]. Un *flot de liens* est un triplet $L = (T, V, E)$ où T est un intervalle de temps, V un ensemble de sommets et E un ensemble de liens temporels. Un lien entre deux sommets u et v sur un intervalle de temps $[b, e]$ est représenté par un lien $(b, e, u, v) \in E$, et donc $E \subseteq T \times T \times V \times V$. Si (b, e, u, v) est un lien, nous appelons $e - b$ la *durée* de ce lien. Les flots de liens que nous utilisons sont non orientés, *i.e.* on ne fait pas de distinction entre $(b, e, u, v) \in E$ et $(b, e, v, u) \in E$. Ils sont également simples, c'est-à-dire qu'il y a au plus un lien entre

deux sommets sur un intervalle de temps donné : pour tout (b, e, u, v) et (b', e', u, v) dans E , $[b, e] \cap [b', e'] = \emptyset$. Il n'y a pas non plus de lien reliant un sommet à lui-même : pour tout (b, e, u, v) dans E , $u \neq v$. Pour éviter toute confusion, nous utilisons le terme *arête* pour les éléments de E_G dans un graphe $G = (V, E_G)$, et le terme de *lien* pour les éléments de E dans un flot de liens $L = (T, V, E)$.

Une *clique d'un flot de liens* est une paire $(C, [t_0, t_1])$ où $t_0, t_1 \in T$, $t_0 < t_1$, sont respectivement appelés le *temps de début* et le *temps de fin* de la clique, et $C \subseteq V$ est l'ensemble des sommets de la clique, avec $|C| \geq 2$. Chaque paire de sommets de C est connectée par un lien qui existe durant tout l'intervalle de temps $[t_0, t_1]$. Formellement, $(C, [t_0, t_1])$ est telle que : pour tout $u, v \in C$ avec $u \neq v$, il existe $(b, e, u, v) \in E$ tel que $[t_0, t_1] \subseteq [b, e]$. Une *k-clique* est une clique contenant k sommets.

Comme pour une clique dans un graphe, une clique dans un flot de liens peut contenir d'autres cliques. Pour éviter les redondances, la notion de maximalité s'applique à la fois en termes de *temps* et en termes de *sommets*. Une *clique maximale en temps* $(C, [t_0, t_1])$ est une clique dont l'intervalle de temps d'existence ne peut pas être étendu : il n'existe aucune clique $(C, [t'_0, t'_1])$ telle que $[t_0, t_1] \subsetneq [t'_0, t'_1]$. Une *clique maximale en sommets* $(C, [t_0, t_1])$ est une clique dont l'ensemble de sommets ne peut pas être étendu : il n'existe aucune clique $(C', [t_0, t_1])$ telle que $C \subsetneq C'$. Enfin, une clique est *maximale* lorsqu'elle est à la fois maximale en temps et maximale en sommets.

Notons que par souci de simplicité, nous utilisons le terme de clique à la fois pour désigner une clique C dans un graphe et une clique $(C, [t_0, t_1])$ dans un flot de liens, alors que ces objets sont de nature différente. Le contexte permettra de lever l'ambiguïté sur le type d'objet considéré.

2.2 Les graphes issus du monde réel

Comme on l'a vu dans l'introduction de cette thèse, les graphes issus du monde réel sont d'une importance capitale pour comprendre la complexité des systèmes qui nous entourent. Lorsque l'on aborde leur analyse, on se retrouve face à une richesse de réseaux, qui émergent dans une multitude de contextes. Une particularité de ces réseaux est qu'ils présentent des caractéristiques qui les distinguent nettement des modèles théoriques abstraits. De plus, comme les graphes étudiés sont de plus en plus massifs, et il n'est pas rare d'avoir à traiter plusieurs millions, voire milliards d'arêtes. Il devient alors nécessaire d'identifier et d'exploiter ces propriétés, afin d'obtenir des algorithmes plus efficaces sur ces cas particuliers de graphes. Par exemple, il est connu que les graphes issus du monde réel possèdent une composante connexe géante, contenant la plupart de leurs sommets. Dans cette section, nous passons en revue une partie de ces caractéristiques, puis nous montrons l'impact qu'elles

peuvent avoir lors du développement d'algorithmes de graphes.

2.2.1 Propriétés des graphes du monde réel

Les réseaux réels présentent généralement une faible densité, c'est-à-dire que le nombre de liens est faible par rapport au nombre de liens maximal que le graphe pourrait contenir. On dit que ces graphes sont *clairsemés*, ou *sparse* en anglais : le nombre moyen de voisins d'un sommet est beaucoup plus faible que le nombre de sommets du graphe. Par exemple, dans un réseau social comme *Facebook*, un compte personnel est en général ami avec quelques centaines d'autres, voire milliers d'autres comptes, ce qui est très faible devant les dizaines de millions d'utilisateurs.

Cette propriété est essentielle dans l'étude des graphes du monde réel, pour lesquels on peut donc faire l'hypothèse que le degré moyen des sommets est très faible par rapport au nombre de sommets n . C'est une caractéristique qui est importante à prendre en compte pour la complexité des algorithmes. Par exemple, cela implique que représenter un graphe du monde réel sous forme de listes d'adjacences est beaucoup plus avantageux en mémoire que sous forme de matrice d'adjacence. En effet, la complexité de stockage en mémoire des listes d'adjacences est en $\Theta\left(\sum_{u \in V} d(u)\right)$, c'est-à-dire en $\Theta(\bar{d} \cdot n)$, où n est le nombre de sommets du graphe et \bar{d} le degré moyen de ses sommets, tandis que la complexité de stockage de la matrice d'adjacence est en $\Theta(n^2)$, ce qui peut être prohibitif pour des graphes à plusieurs centaines de milliers, voire millions de sommets. La première complexité est bien meilleure en pratique du fait que $\bar{d} \ll n$. En revanche, tester si une arête $\{u, v\}$ existe est plus coûteux avec les listes d'adjacence, car il faut parcourir la liste des voisins de u , ou celle de v , et cela se fait en $\Theta(\min(d(u), d(v))) = \Theta(d)$, tandis que pour la matrice d'adjacence, si $u = u_i$ et $v = u_j$, il suffit de regarder la case de la matrice M_{ij} , en $\Theta(1)$. Néanmoins, du fait de la faible densité, le parcours de la liste d'adjacence d'un sommet reste une opération à faible coût en moyenne. Dans toute cette thèse, pour ces raisons, *nous travaillons systématiquement avec des listes d'adjacence*.

Les graphes du monde réel ont également comme caractéristique particulière d'avoir une distribution de degrés hétérogène. Certains noeuds, appelés *hubs*, sont connectés à un très grand nombre d'autres noeuds, quand la majorité n'est connectée qu'à peu d'autres. Ces hubs jouent un rôle clé dans la connectivité du réseau et sont souvent associés à des entités importantes ou influentes dans le système étudié. Cela peut correspondre par exemple à des profils de célébrités sur un réseau social en ligne. Ainsi, le degré maximal dans ces graphes peut être élevé, mais la valeur de la dégénérescence reste faible en pratique.

Un autre aspect intéressant des graphes du monde réel est leur propriété de transitivité. Cela signifie que si deux noeuds u et v sont connectés à un même noeud w , alors il y a une forte probabilité que ces noeuds u et v soient également liés entre

eux. De manière générale, si deux sommets ont des voisins en commun, ces voisins ont de fortes chances d'être connectés entre eux. Par exemple, dans le réseau des interactions sociales, deux de mes amis ont beaucoup de chances de se connaître entre eux, tandis que deux personnes prises au hasard ont beaucoup de chances de ne pas se connaître. Cette propriété favorise la formation de triangles, et permet de prédire ou de valider la résistance d'un lien dans le réseau, ce qui rend le graphe robuste par exemple face aux erreurs potentielles lors de son enregistrement expérimental. [LK03 ; LZ11].

De manière notable, même si ces réseaux sont souvent caractérisés par leur caractère clairsemé, on constate que des structures localement denses existent, avec des sous-graphes très denses, bien qu'elles restent relativement petites en taille par rapport au nombre de sommets du graphe. Les graphes du monde réel présentent ainsi une forte tendance à former des *clusters*, ou *communautés*. Les communautés sont des ensembles de sommets, et au sein de ces communautés, les nœuds sont fortement interconnectés, tandis que les liens entre les communautés sont moins nombreux. Cette organisation reflète la présence de groupes cohérents au sein du réseau réel. La détection de ces communautés est essentielle pour comprendre la structure et le fonctionnement du système étudié [GN02 ; For10]. Nous revenons sur cette notion très importante dans la section 2.4.3, ainsi que dans les chapitres 3 et 6.

Enfin, les interactions du monde réel s'inscrivent dans le temps. Les étudier implique donc de tenir compte de leur dynamique. Nous détaillons ce point dans la section 2.3.

2.2.2 Expression des complexités avec des variables pertinentes

Les propriétés spécifiques des réseaux du monde réel peuvent être exploitées pour concevoir des algorithmes qui s'exécutent plus rapidement sur des données réelles par rapport à ce qui est prévu par l'analyse de leur pire cas. En ce sens, on peut chercher à exprimer la complexité des algorithmes par rapport à plusieurs paramètres qui correspondent à des propriétés particulières de l'entrée ou de la sortie. Cette méthode fournit une description plus fine que dans le cadre classique, pour évaluer la complexité d'un algorithme sur des entrées spécifiques. Elle est particulièrement pertinente, car les performances des algorithmes peuvent différer considérablement et peuvent être irréguliers en fonction des instances d'entrée. Par exemple, si on identifie une famille d'entrées qui déclenche un temps d'exécution exponentiel, l'algorithme sera catégorisé comme au moins exponentiel dans le pire cas, et ce même si toutes les instances rencontrées dans la pratique sont traitées en temps linéaire (bien que cela puisse être modéré par une complexité en moyenne). Utiliser des paramètres adaptés pour exprimer la complexité permet d'aborder ces différences de comportement du temps de calcul d'un point de vue théorique et peut expliquer pourquoi certains

algorithmes sont si performants dans les cas rencontrés en pratique, par rapport aux pires cas théoriques.

Par exemple, la dégénérescence est un bon facteur pour quantifier le caractère localement dense d'un graphe. En effet, si un graphe a une faible dégénérescence, cela signifie que si un sommet a beaucoup de voisins, la plupart ne sont pas voisins entre eux, et donc il n'y a pas de gros ensembles de sommets densément connectés. Cette information est moins évidente à déduire à partir du degré maximal ou du degré moyen du graphe. De même, si un sommet a une dégénérescence élevée, c'est qu'il a beaucoup de voisins qui sont voisins entre eux, et alors ce sommet se trouve dans une zone locale dense du graphe, tandis qu'un sommet avec un fort degré peut n'être connecté qu'à des sommets de faible degré, et donc ne pas se trouver dans une zone particulièrement dense du graphe. Cela a permis de développer des complexités théoriques utiles exprimées à l'aide de la dégénérescence. Par exemple, la dégénérescence s'est avérée utile pour obtenir un algorithme efficace d'énumération de k -cliques [DBS18]. Dans ce travail, les auteurs fournissent un algorithme pour énumérer l'ensemble des k -cliques en $\mathcal{O}\left(m \cdot \left(\frac{c}{2}\right)^{k-2}\right)$, où m est le nombre d'arêtes et c la dégénérescence. Bien que ce problème soit polynomial, pour des graphes avec de petites valeurs de c , comme c'est le cas des graphes du monde réel, cette complexité montre d'un point de vue théorique que le nombre de calculs effectivement réalisés est faible, rendant l'algorithme extrêmement efficace sur les instances du monde réel.

Enfin, le tri des noeuds d'un graphe dans un ordre pertinent s'est avéré être une sous-routine clé pour résoudre de nombreux problèmes dans les graphes massifs du monde réel. Sans être exhaustif, on peut citer par exemple des problèmes du domaine de l'énumération de sous-graphes connexes dans les graphes massifs, avec la recherche d'un sous-graphe dense [DCS17], l'énumération des triangles [Léc+23 ; Lat08], l'énumération des cliques maximales [ES11] ou la recherche d'une clique maximum [Ros+14]. Dans ces travaux, l'ordre des sommets permet de raffiner la complexité des algorithmes étudiés, en faisant apparaître des paramètres qui ont spécifiquement une valeur faible dans le cas pratique des graphes issus du monde réel. Il permet par exemple de changer la complexité de l'énumération des k -cliques, en la faisant passer de $\mathcal{O}\left(m \cdot \left(\frac{d}{2}\right)^{k-2}\right)$ à $\mathcal{O}\left(m \cdot \left(\frac{c}{2}\right)^{k-2}\right)$, où d est le degré maximal et c la dégénérescence du graphe [DBS18]. Or, dans les graphes du monde réel, la dégénérescence est en général beaucoup plus faible que le degré maximal, ce qui rend cette complexité bien meilleure en pratique. Tirer profit de l'ordre des sommets est d'autant plus pertinent que beaucoup d'ordres peuvent être obtenus en un temps quasi-linéaire. Par exemple, le tri des noeuds en fonction de leur degré peut être effectué en calculant d'abord le degré de chaque noeud en $\mathcal{O}(m + n)$, puis en les triant en $\mathcal{O}((n \cdot \log(n)))$. De même, trier les sommets par dégénérescence peut être obtenu en temps linéaire par un algorithme qui supprime de manière répétée un noeud de degré minimum [BZ03]. Nous reviendrons dans le chapitre 4 sur l'impact

que cet ordre a pour l'énumération des cliques maximales dans les graphes et dans les graphes bipartis.

2.2.3 Valider les algorithmes par une étude expérimentale

Étudier un algorithme par sa complexité n'est pas toujours suffisant pour expliquer la rapidité de son exécution sur les réseaux du monde réel, même en faisant apparaître des paramètres pour tenir compte des propriétés spécifiques des graphes utilisés. En fait, si on souhaite améliorer l'efficacité en pratique, c'est le temps d'exécution qui importe le plus. Dans ce cas, il est donc intéressant de développer une conception des algorithmes adaptée aux données rencontrées en pratique plutôt que de chercher à améliorer la complexité théorique dans le pire cas. Par exemple, l'algorithme de Danisch *et al.* [DCS17] proposé pour trouver des sous-graphes denses a un temps d'exécution cubique dans le pire des cas, alors qu'il se comporte en pratique comme un algorithme en temps linéaire dans les expériences sur les graphes du monde réel. De même, la complexité de l'algorithme pour énumérer toutes les cliques maximales d'un graphe proposé par Eppstein *et al.* [ES11] est en $\mathcal{O}(n \cdot 3^{c/3})$. Or, cet algorithme permet de traiter efficacement des graphes avec des dégénérescences c valant plusieurs centaines [CT22], ce qui ne peut pas être expliqué par cette formule de complexité.

Ainsi, il est essentiel d'associer à chaque algorithme une étude expérimentale, pour montrer leur efficacité directement sur les instances du monde réel. C'est la démarche que nous adoptons dans cette thèse, où, bien que nous fournissons les complexités de nos algorithmes, ce sont les intuitions et les études expérimentales qui valident leur gain d'efficacité par rapport à l'état de l'art auquel nous nous comparons.

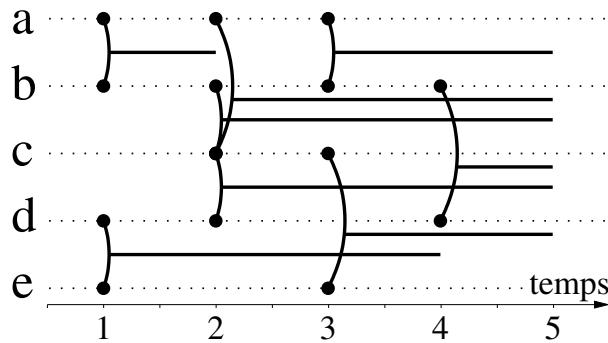
2.3 Modéliser les réseaux temporels

Les réseaux temporels trouvent leur utilité dans divers contextes, comme le montrent Holme et Saramäki dans leur livre très complet sur le sujet [HS12]. Par exemple, l'analyse des communications électroniques constitue l'une des applications majeures de ces réseaux [BDK15 ; LH08 ; Can+08 ; Gau+16]. Ces données, qu'elles proviennent d'échanges de courriels, de SMS, de messages instantanés ou d'appels téléphoniques, requièrent une analyse temporelle, notamment pour comprendre la diffusion d'informations ou de virus informatiques, et par exemple comprendre comment contenir la propagation d'une fausse information ou de logiciels malveillants [Kar+11 ; Tan+11]. En outre, les interactions physiques entre individus revêtent une grande importance, pour étudier par exemple la propagation des maladies infectieuses et la dynamique des épidémies [Ise+11 ; Ste+11 ; MCF14]. Jusqu'à récemment, ces réseaux de proximité ou de contacts étaient difficiles à étudier

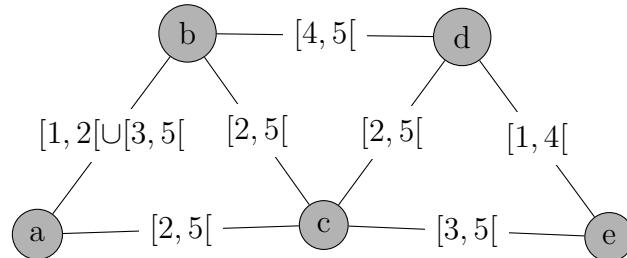
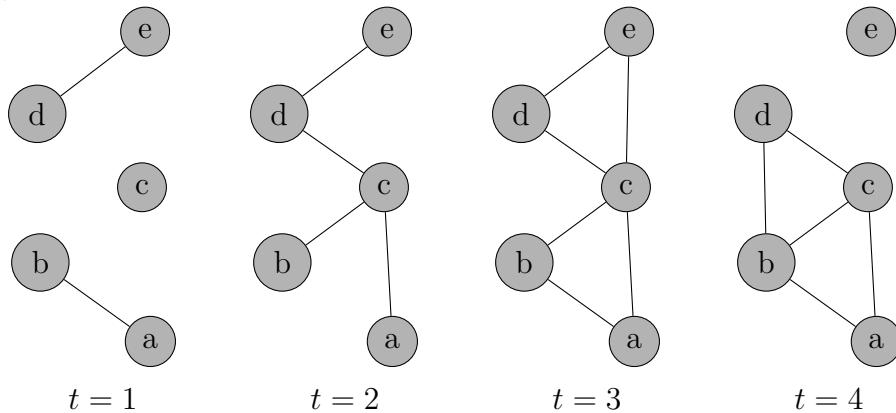
à grande échelle, car cela nécessitait des travaux de terrain fastidieux. Cependant, grâce à des dispositifs Bluetooth ou autres appareils électroniques, il est désormais possible de collecter ces données plus facilement [EP06 ; Cat+10]. Dans un tout autre domaine, en biologie cellulaire, les réseaux temporels trouvent leur place dans l'étude de l'interactome, qui représente l'ensemble des interactions moléculaires dans une cellule [PSS10]. En effet, bien que traditionnellement représentées sous forme de graphes statiques, ces interactions sont souvent activées de manière intermittente. On peut également citer les réseaux écologiques, qui décrivent les interactions entre espèces. Ils varient en fonction des saisons, des cycles de vie des organismes et de l'évolution [JF16 ; Ula04]. Nous pourrions encore citer beaucoup d'autres exemples, comme les réseaux sociaux dont la topologie change entre un groupe d'individus au fur et à mesure que les relations sociales évoluent [Zha+12], ou le trafic internet [Har+16 ; VL14], le déplacement du bétail [DEV14 ; PTL19], les réseaux de transport où les éléments se déplacent sur des itinéraires... Dans chaque contexte pratique, les chercheurs et les ingénieurs sont confrontés au défi d'analyser la nature conjointement temporelle et structurelle des interactions, et ils développent des méthodes et des outils spécifiques pour le faire.

En raison de leur polyvalence, les graphes temporels ont été introduits dans de nombreuses disciplines de recherche, indépendamment et en parallèle, et outre les différentes conventions de dénomination, il existe également différents formalismes des graphes temporels. Nous nous intéressons ici aux graphes temporels dans lesquels les arêtes existent dans des intervalles de temps continus. La figure 2.1 présente les trois modèles principaux que nous considérons, et que nous discutons ci-dessous.

Une approche courante consiste à modéliser un réseau dynamique par une séquence de graphes : le temps est découpé en intervalles, et pour chacun un graphe est construit en agrégant les interactions qui existent sur cet intervalle [Blo+12 ; Li+17 ; LKF07]. Ces graphes sont généralement appelés *graphes instantanés*, et un exemple en est donné dans la figure 2.1 (c). Cela permet d'utiliser directement les outils classiques de graphes, pour en suivre l'évolution temporelle d'un graphe instantané à l'autre. Chaque arête d'un graphe instantané est perçue comme existant sur l'ensemble de l'intervalle pour lequel ce graphe est défini. Or, cela soulève des questions sur l'existence d'une fenêtre temporelle d'analyse adéquate. En effet, les intervalles d'agrégation doivent être suffisamment longs pour capturer un ensemble significatif d'interactions. Mais ils ne doivent pas non plus être trop longs, pour ne pas perdre trop d'information concernant la dynamique temporelle, car toutes les interactions sont fusionnées au sein d'un même intervalle. En effet, des propriétés sur ces graphes peuvent mener à de mauvaises interprétations. Par exemple, il peut exister un chemin entre deux sommets dans un graphe instantané, sans pour autant que les sommets soient connectés d'un point de vue temporel, c'est-à-dire que les liens du chemin soient rangés par ordre chronologique. Le choix des fenêtres de temps d'analyse pertinentes n'est donc pas évident et a conduit à des travaux



(a) Représentation du réseau temporel sous forme de flot de liens.

(b) Représentation du réseau temporel sous forme de *Time Varying Graph*.

(c) Représentation de réseau temporel sous forme de graphes instantanés.

FIGURE 2.1 – Trois formalismes de graphes temporels, dans lesquels les arêtes existent sur des intervalles de temps continus : flots de liens, *Time Varying Graph* et graphes instantanés. Chacune de ces trois figures représente le même réseau temporel.

spécifiques [HLS09 ; LCF19 ; RPB13 ; Kri+12 ; SWG16].

Pour pallier ces limites, plusieurs auteurs proposent de modéliser les interactions temporelles par des formalismes qui prennent en compte explicitement le temps. Parmi eux, l'un des plus courants est celui des *Time Varying Graphs (TVG)*, qui représentent un réseau dynamique sous la forme d'un graphe où les liens ont des attributs correspondant aux dates auxquelles ils existent dans le réseau [San+11 ; Cas+12]. Un exemple de TVG est donné dans la figure 2.1 (b). Cela permet d'exprimer directement la structure des interactions par le graphe agrégé de tous les liens et la dynamique temporelle sous-jacente grâce à l'étiquetage des arêtes. Le but de ces travaux a été de proposer un formalisme qui intègre et unifie les concepts existants dans la littérature, permettant d'exprimer des concepts communs aux différents domaines (chemins temporels, distance, ...) tout en gardant leurs spécificités.

Néanmoins, de nombreux autres formalismes existent, comme celui de considérer les graphes temporels comme un graphe qui subit des mises à jour, avec des arêtes et sommets qui sont créées ou supprimées au fur et à mesure du temps [DFI18 ; Bou+18 ; DST19]. Ce formalisme implique de développer des méthodes algorithmiques de mise à jour efficace des propriétés du graphe étudiées, ce qui peut être assez coûteux, tout en l'étant moins que d'utiliser une méthode globale à chaque mise à jour. Nous reviendrons sur cette comparaison dans le chapitre 6, avec la méthode de détection de communautés dynamiques DCPM [PBV07 ; Bou+18]. Kosztakos définit en 2009 un graphe temporel en recopiant chaque nœud aux instants où il existe un lien contenant ce nœud, et des liens sont ajoutés pour connecter une copie d'un nœud à son successeur dans le temps, s'il en a un [Kos09]. Ce formalisme a été utilisé par exemple pour étudier des notions de centralités temporelles [KA12]. Il y a également des travaux où les graphes temporels sont représentés comme des réseaux multi-couches [Yu+20 ; WFZ16]. D'autres représentations, souvent appelées simplement *réseaux temporels* [Hol15], visent à rassembler les différents modèles de systèmes dynamiques en interaction, sans faire un choix strict de formalisme.

Dans cette thèse, nous utilisons le formalisme des flots de liens, qui associe un intervalle de temps à chaque interaction [LVM18], nous en avons donné la définition dans la section 2.1, et un exemple de flot de liens est illustré par la figure 2.1 (a). Il s'agit d'un formalisme qui permet de rendre compte de l'aspect temporel des données sans paramètre ni choix restrictif d'échelle de temps, et qui permet de donner un rôle symétrique à l'aspect structurel et temporel des interactions. Nous verrons que cette conception nous permet de développer des intuitions particulières pour le développement de nos algorithmes : nous nous servons de ce formalisme pour adapter des concepts existants dans l'analyse de réseaux complexes pour des contextes dynamiques. Notez néanmoins que ce formalisme est équivalent à celui des TVG (dans le cas où les objets considérés sont finis), ou d'un graphe mis à jour, et qu'il est simple de passer d'un mode de représentation à un autre.

2.4 Description de la structure locale des graphes

L'étude des graphes du monde réel passe par une étude de leur structure, afin de comprendre la manière dont les interactions sont organisées. On peut par exemple catégoriser ou résumer leur structure, ainsi qu'isoler des ensembles de sommets particuliers à analyser. À cette fin, nous présentons ici les problèmes généraux autour de la détection de motifs (section 2.4.1), des sous-graphes denses en général (section 2.4.2), et des communautés (section 2.4.3).

2.4.1 Motifs

Les motifs représentent des sous-structures récurrentes ou significatives présentes dans les graphes et leur identification est importante pour comprendre les propriétés des réseaux complexes. Ce sont des sous-graphes connexes de petite taille (souvent cinq noeuds ou moins) qui se produisent plus ou moins fréquemment que prévu par le hasard, révélant ainsi des sous-structures qui jouent un rôle majeur dans le fonctionnement du réseau [AMS96].

L'énumération de motifs dans les graphes est d'une grande importance pour plusieurs raisons. Tout d'abord, l'abondance de certains types de motifs est caractéristique de familles de graphes et peut donc être révélatrice des mécanismes de formation du réseau [Mil+02]. Ils peuvent correspondre à des propriétés ou des régions du graphe impliquées dans des processus spécifiques, et permettent de caractériser la structure des réseaux du monde réel [Alo07]. En analyse de réseaux sociaux, les motifs sont fréquemment recherchés pour repérer des types de configurations sociales particuliers [WF94]. Par exemple, si on observe une étoile dans le réseau, qui est un motif où un sommet est relié à d'autres sommets n'ayant aucune arête entre eux, on en déduit que les individus ne peuvent communiquer que par le noeud central, qui est ainsi dans une situation de pouvoir [CP19]. La recherche de motifs est utile en biologie, où ils sont en général appelés *graphlets* [Prž07]. Notamment, une revue exhaustive sur une variété d'outils et d'algorithmes pour la découverte de motifs dans les réseaux biologiques a été réalisée en 2012 [Won+12], dans laquelle les auteurs mettent en évidence l'importance des motifs dans la compréhension des fonctions biologiques des réseaux : chaque motif significatif est présent pour une raison et a une fonction dans le réseau, comme un motif de gènes qui interagissent entre eux en modifiant l'expression d'autres gènes [MA03]. De plus, l'énumération de motifs facilite la comparaison de différents graphes en identifiant des motifs similaires, ce qui permet d'appliquer des méthodes d'apprentissage [Szc17; Zel+18]. Plus généralement, l'utilisation des motifs comme caractéristiques d'entrée d'algorithme d'apprentissage sur graphes s'est récemment répandu, en raison de leur importance dans le fonctionnement des réseaux [Ham20].

Au fil des années, de nombreuses approches ont été développées pour l'énumération de motifs dans les graphes. Dans ce contexte, plusieurs revues et études ont

été publiées pour examiner les différents aspects de cette problématique. En 2012, Masoudi-Nejad *et al.* ont réalisé une revue axée sur les aspects computationnels des principaux algorithmes de découverte de motifs de graphes, en mettant en évidence les avantages et les limitations de ces algorithmes [MSK12]. En effet, la détection de motifs dans les graphes massifs est complexe en raison de l'augmentation exponentielle du nombre de sous-graphes possibles lorsqu'on augmente la taille du graphe ou du motif [Rib+21]. Ainsi, les approches exhaustives qui consistent à énumérer ou compter tous les motifs possibles dans un graphe deviennent rapidement inefficaces pour les graphes de grande taille, pour lesquels on ne peut travailler que sur de petits motifs [Ahm+15 ; PSV17].

Dans les graphes temporels, la recherche de motifs permet également de caractériser la manière dont ils sont organisés, pour en extraire de l'information [Kov+13 ; Liu+22]. Plusieurs études ont été réalisées pour explorer ces motifs. Par exemple, Panzarasa *et al.* [POC09a] ont examiné les comportements et les interactions sociales des utilisateurs dans une communauté en ligne, en étudiant les motifs dans les interactions dynamiques, avant que le concept de *motif temporel* ne soit introduit formellement par Kovanen *et al.* en 2011 [Kov+11] pour identifier des séquences d'événements similaires dans les réseaux dépendant du temps. En 2017, Paramjape *et al.* [PBL17] ont proposé une méthodologie pour compter efficacement les motifs de graphe temporel dans des réseaux réels, soulignant les différences significatives entre les fréquences de motifs de différents domaines. Ces études ouvrent de nouvelles perspectives sur l'analyse des graphes temporels, y compris pour l'étude des réseaux temporels massifs [Gao+22].

2.4.2 Sous-graphes denses, k -cliques, k -plexes

Parmi les motifs d'un graphe, les motifs denses suscitent particulièrement l'intérêt [Lee+10 ; DCS17]. L'objectif est d'identifier des sous-ensembles de noeuds (sous-graphes) qui ont une densité de connexions élevée entre eux [GKT05 ; Fra+06]. La recherche de sous-graphes denses présente des défis algorithmiques intéressants, en raison de sa complexité; certains problèmes de recherche de sous-graphes denses sont NP-complets [AC09].

Les cliques sont les sous-graphes les plus denses qui soient. En 1985, Chiba et Nishizeki ont introduit une méthode d'énumération des k -cliques [CN85], dans laquelle la fonction qui énumère les k -cliques sélectionne un sommet et énumère récursivement l'ensemble des $(k - 1)$ -cliques du sous-graphe induit par les voisins de ce sommet, puis elle supprime ce sommet du graphe et réitère l'opération jusqu'à épuisement de l'ensemble de sommets. En 2018, Danisch *et al.* ont revisité cet algorithme pour tenir compte de l'ordre des sommets traités; ils fournissent alors l'une des meilleures implémentations actuellement disponibles pour la résolution de ce problème [DBS18]. Cette approche permet de lister toutes les k -cliques pour

de petites valeurs de k sur des graphes du monde réel contenant jusqu'à plusieurs milliards d'arêtes, en seulement quelques heures. En 2020, Jain et Seshadhri. ont présenté l'algorithme *Pivoter*, qui compte exactement le nombre de k -cliques pour toutes les valeurs de k , en utilisant la construction d'une structure appelée *Succinct Clique Tree* [JS20]. Cette structure stocke une représentation unique compressée de toutes les cliques du graphe, permettant ainsi d'obtenir des comptes précis pour des graphes de grande taille. L'énumération des cliques maximales est également une tâche importante. C'est un problème NP-difficile, là où l'énumération des k -cliques est polynomiale. C'est donc un problème plus complexe à traiter, sur lequel nous revenons dans le chapitre 4, dans lequel nous établissons un état de l'art détaillé des meilleurs algorithmes permettant de résoudre ce problème sur les graphes du monde réel, qui se basent sur le célèbre algorithme de Bron-Kerbosch [BK73].

Parmi les applications de l'énumération des k -cliques, on trouve le *k -clique densest subgraph problem* [Tso15] qui généralise le problème bien étudié du sous-graphe de densité maximum [Gol84]. Ce problème est particulièrement intéressant, car il a été constaté que, par rapport au problème du sous-graphe de densité maximum, cette méthode permet de mieux détecter les quasi-cliques de grande taille dans les réseaux, où une quasi-clique est un ensemble de sommets avec une densité dépassant un certain seuil fixé [Tso+13]. Des méthodes efficaces ont récemment été proposées pour résoudre ce problème [Sun+20]. On peut également citer la détection de communauté par percolation de cliques, appelée *Clique Percolation Method (CPM)* sur laquelle nous travaillons dans les chapitres 3 et 6, et qui permet de calculer des communautés qui se chevauchent, dont nous discutons un peu plus tard dans cette section.

Il existe plusieurs relaxations à la définition des cliques, dont on peut avoir un aperçu dans la revue de Patillo *et al.* [PYB13]. Parmi elles, les k -plexes, introduits par Seidman et Foster en 1978 [SF78], sont une des relaxations les plus couramment utilisées. Un k -plex est un sous-graphe dans lequel chaque noeud est connecté à tous les autres noeuds du sous-graphe, avec une tolérance de k arêtes manquantes. Cette flexibilité les rend plus adaptés à certains cas pratiques, car des arêtes peuvent manquer lors de l'enregistrement des réseaux, ou bien des regroupements denses intéressants peuvent ne pas former des cliques parfaites. Plusieurs approches algorithmiques ont été développées pour résoudre efficacement le problème de l'énumération des k -plexes dans les grands graphes. Parmi celles-ci, Balasundaram *et al.* ont introduit le problème du *maximum k -plex* [BBH11] ; ils établissent la NP-complétude du problème et fournissent un algorithme d'énumération pour les graphes issus du monde réel. En 2017, Conte *et al.* ont proposé une nouvelle approche pour énumérer de grands k -plexes dans les graphes, accélérant la recherche de plusieurs ordres de grandeur, en exploitant des méthodes de réduction de l'espace de recherche et des techniques efficaces pour le calcul des cliques maximales [Con+17]. En 2018, ils améliorent leur méthode en présentant *D2K*, un algorithme capable de trouver

rapidement de grands k -plexes dans de très grands graphes en seulement quelques minutes, en exploitant la propriété de faible dégénérescence des graphes issus du monde réel et le fait que les k -plexes de taille suffisamment grande ont un diamètre de 2 [Con+18]. Enfin, en 2020, Zhou *et al.* ont développé *FaPlexen*, un algorithme d'énumération de tous les k -plexes maximaux, en fournissant une garantie de complexité dans le pire cas [Zho+20]. On peut également utiliser les k -plex comme outil de prédiction de liens, car les arêtes manquantes sont probablement de bons candidats pour les liens manquants dans les réseaux. Il existe d'autres relaxations des cliques, de nature différente, comme la quasi-clique que nous avons mentionnée où la densité du sous-graphe induit doit être supérieure à un paramètre, ou encore le k -club dont l'ensemble des sommets doit avoir un diamètre inférieur à k .

Dans les réseaux dynamiques, la question de l'énumération des cliques est apparue relativement récemment comme une question de recherche importante pour décrire la structure des données d'interactions temporelles. Ce problème a été envisagé de différentes manières. Dans certains cas où un réseau dynamique est considéré comme une séquence de graphes instantanés, cela conduit à aborder le problème sous l'angle de l'évolution des cliques dans les graphes au cours du temps. C'est par exemple le point de vue adopté par Sun *et al.* [Sun+17], ou encore Das *et al.* [DST19]. D'autres travaux modélisent le réseau dynamique comme un flot de liens, reconnaissant ainsi l'aspect intrinsèquement temporel de la clique elle-même et la considèrent donc comme un objet qui devrait être redéfini dans le contexte dynamique [VLM16 ; Him+16 ; Him+17 ; VML18 ; Ben+19]. Ce dernier point est abordé en détail dans le chapitre 5. Bentert *et al.* [Ben+19] ont été les premiers à étudier l'énumération des k -plexes dans les graphes temporels, en adaptant l'énumération des cliques maximales dans les flots de liens proposée par Himmel *et al.* [Him+17], et en s'inspirant des adaptations de l'algorithme de Bron-Kerbosch à l'énumération des k -plexes qui existent pour les graphes statiques [Con+18 ; WP07].

2.4.3 Détection de communautés

Comme on l'a vu précédemment, l'étude des graphes issus du monde réel met en évidence des propriétés modulaires : ils contiennent des communautés de noeuds qui sont densément connectés entre eux, et peu avec les autres. Ce qui est intéressant dans l'étude de ces communautés, c'est qu'elles sont souvent liées à des propriétés fonctionnelles du système étudié, par exemple des groupes d'individus qui interagissent ensemble dans la société [SC11 ; Eve+11], ou des modules fonctionnels dans les réseaux biologiques [Rav+02 ; PL06]. Il existe un grand nombre de méthodes de détection de communautés. De nombreuses définitions ont été élaborées et nous n'avons pas l'intention dans faire une revue exhaustive ici. On peut néanmoins différencier deux types de regroupement en communautés : la partition des sommets dans les graphes [FC12], et les communautés qui se chevauchent [XKS13].

Fortunato et Castellano ont proposé en 2012 une typologie pertinente des méthodes traditionnelles de recherche de communautés en partition des sommets [FC12]. Nous reprenons ici leur revue, en donnant quelques exemples de références classiques de la littérature et les limites associées à ces méthodes. Premièrement, le problème qu'ils appellent *graph partitionning* consiste à diviser les sommets d'un graphe en un nombre fixé de communautés de taille prédéfinie, de manière à minimiser la coupe, c'est-à-dire le nombre d'arêtes entre ces groupes [Pot97; KL70]. Il est à noter qu'il est important de spécifier le nombre de communautés dans la partition, car sans cette spécification, la solution serait triviale avec tous les sommets dans le même groupe, donnant une taille de coupe nulle. De même, il est nécessaire de prendre en compte la taille des communautés, sinon la solution pourrait simplement séparer le sommet de degré le plus bas du reste du graphe, ce qui serait peu intéressant. Or, spécifier ces valeurs n'est pas toujours possible, car la structure communautaire d'un graphe est souvent inconnue. Le *spectral clustering* est une classe de méthodes qui utilise les vecteurs propres de matrices particulières pour regrouper les sommets [DH73; Fie73]. Le principe consiste, à partir d'une matrice sur les sommets du graphe, à partitionner les sommets en des espaces de plus petite dimension. Il existe plusieurs types de matrices à utiliser, la plus courante est la matrice laplacienne qui est la matrice des degrés à laquelle on soustrait la matrice d'adjacence du graphe. Le calcul des vecteurs propres peut être coûteux pour les graphes de grande taille, mais il existe des techniques d'approximation pour résoudre ce problème. Le regroupement spectral peut également aider à estimer le nombre de communautés en examinant les écarts significatifs entre les valeurs propres. Enfin, il existe aussi des algorithmes *divisifs*, qui détectent des arêtes reliant les sommets de différents groupes et les suppriment, ce qui conduit à une division en communautés. L'algorithme fondamental pour résoudre ce problème est celui de Girvan et Newman [NG04].

Cependant, on peut trouver plus pertinent de trouver des communautés qui se chevauchent, c'est-à-dire qu'un nœud peut appartenir à plusieurs communautés. Dans un réseau social par exemple, il est clair que chaque personne appartient à plusieurs communautés, qui peuvent correspondre à ses loisirs, sa famille, son entourage professionnel, *etc.* Nous en donnons une illustration dans la figure 2.2, qui présente un exemple de toutes les communautés auxquelles une personne peut appartenir. Il en va de même dans d'autres réseaux complexes, tels que les réseaux biologiques, où un nœud peut avoir plusieurs fonctions. Il a en effet été montré que le chevauchement est une caractéristique importante de nombreux réseaux du monde réel [Pal+05; Kel+12].

Xie *et al.* [XKS13] ont proposé une revue complète des algorithmes de détection de communautés qui se chevauchent. Ils comparent différentes approches en termes de performances de détection des communautés et proposent un cadre pour évaluer la capacité des algorithmes à détecter des communautés qui se recouvrent beaucoup ou

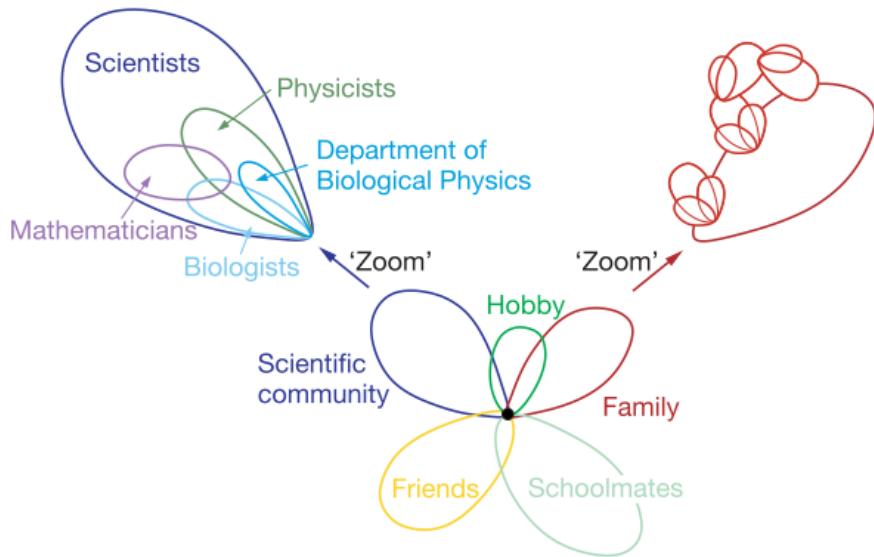


FIGURE 2.2 – Illustration du concept de communautés qui se chevauchent, d’après Palla et al. [Pal+05]. Le point central correspond à une personne, et ses communautés sont représentées. Elles se recoupent et certaines sont incluses dans d’autres ; nous voyons par exemple que la communauté *Scientists* contient plusieurs sous-communautés selon la discipline.

non. Leurs résultats montrent que les algorithmes de détection de communautés qui se chevauchent peuvent être efficaces, mais qu’il reste des défis à relever, notamment pour les réseaux présentant une forte densité et diversité de chevauchement. Dans cette revue, les auteurs examinent la détection de communautés dans les réseaux complexes en se penchant sur cinq catégories de communautés, que nous détaillons dans la suite de ce paragraphe. Une méthode couramment utilisée pour détecter de telles communautés est la percolation des cliques (CPM), introduits par Palla *et al.* en 2005 [Pal+05], que nous présentons dans le chapitre 3. Elle repose sur l’idée que les communautés sont formées d’ensembles de sous-graphes entièrement connectés qui se recouvrent. Une autre approche explore le partitionnement des liens plutôt que des noeuds pour détecter la structure communautaire. Un noeud peut alors appartenir à plusieurs communautés si ses liens sont répartis dans plusieurs communautés [ABL10 ; EL10]. D’autres algorithmes font grossir les communautés de manière naturelle, à l’aide d’une fonction de bénéfice qui caractérise la qualité des groupes de noeuds densément connectés, pour voir s’ils sont à étendre ou non [Bau+05 ; LFK09]. Les algorithmes de *fuzzy detection* construisent les communautés en minimisant une fonction objectif entre les noeuds et les communautés [Nep+08 ; ZWZ07]. Enfin, l’algorithme de propagation des étiquettes attribue des étiquettes aux noeuds et les fait évoluer jusqu’à un équilibre, et à la fin les sommets ayant une même étiquette appartiennent à la même communauté. Il a été étendu à la détection de communautés qui se chevauchent, en autorisant un noeud à avoir plusieurs étiquettes [XS11 ; Che+10].

De même, l'étude des communautés dans les graphes temporels a reçu beaucoup d'attention, comme le montrent deux revues récentes sur le sujet [Dak+19 ; AR22]. Une approche standard consiste à considérer les graphes temporels comme une séquence de graphes instantanés et à exécuter des algorithmes de détection de communautés de graphes sur chacun d'entre eux. Il est alors logique de faire correspondre les communautés obtenues d'un pas de temps à l'autre afin d'obtenir des groupes cohérents dans le temps [AG10]. Cette stratégie a été étudiée pour diverses applications, comme les réseaux de communication mobile [Ngu+11] ou l'analyse des réseaux sociaux [Ros+17]. Cependant, l'instabilité de nombreuses méthodes de détection des communautés rend cette tâche difficile à réaliser correctement [CRA17]. En outre, certains travaux soulignent l'importance de réaliser une détection de communautés à la volée, auquel cas les communautés sont mises à jour à chaque pas de temps, en intégrant les changements du graphe aux communautés existantes [Cui+13 ; Pan+14]. Si la méthode est suffisamment rapide, il est possible de réaliser une analyse des communautés à la volée des données. Toutefois, ce type de méthode a souvent pour inconvénient de perdre une partie du sens qu'une communauté peut avoir à long terme. Dans le chapitre 6, nous fournissons un nouvel algorithme de détection de communautés dans les flots de liens, qui vise à dépasser ces limites.

Percolation de cliques efficace en mémoire pour détecter des communautés

Résumé

La détection automatique de groupes de noeuds pertinents dans les graphes massifs issus du monde réel, c'est-à-dire la détection de communautés, a des applications dans de nombreux domaines et a fait l'objet d'une grande attention au cours des vingt dernières années. La méthode par percolation de cliques (CPM pour *Clique Percolation Method*) est une méthode courante conçue pour trouver des communautés qui se chevauchent (c'est-à-dire telles qu'un noeud peut appartenir à plusieurs communautés). Cette méthode formalise la notion de communauté comme une union maximale de k -cliques qui peuvent être atteintes l'une par rapport à l'autre par une série de k -cliques adjacentes, où deux k -cliques sont adjacentes lorsqu'elles ont $k - 1$ noeuds en commun. Malgré de nombreux efforts, CPM n'a pas pu être adapté pour passer à l'échelle de graphes massifs, même pour de petites valeurs de k .

Des travaux récents ont montré qu'il est possible d'énumérer efficacement toutes les k -cliques dans de très grands graphes issus du monde réel pour les petites valeurs de k (typiquement $k \leq 10$). Nous nous appuyons sur ces travaux pour améliorer l'implémentation de l'algorithme de percolation de cliques CPM, afin d'appliquer cette détection de communautés à des graphes plus massifs que ce qui se fait dans l'état de l'art. Cette implémentation est confrontée à des limites de mémoire. Nous proposons alors un nouvel algorithme, appelé CPMZ, qui fournit une solution proche de la solution exacte fournie par CPM, qui nécessite plus de temps de calcul, mais moins de mémoire.

3.1 Introduction

Le problème de la détection des communautés dans les réseaux réels a fait l'objet d'une grande attention ces dernières années. Comme on l'a vu dans le chapitre 2

sur les préliminaires, de nombreuses définitions de communautés existent, qui correspondent à différentes exigences sur le type de communautés à détecter et/ou sur certaines propriétés du graphe étudié. Chaque définition peu en effet dépendre de propriétés différentes du graphe, locales ou globales, les communautés peuvent être des partitions ou des ensembles qui se chevauchent, elles peuvent avoir une structure hiérarchique, les liens peuvent être pondérés, *etc.* Dans la pratique, des algorithmes doivent être conçus pour extraire ces communautés dans des grands graphes.

Dans ce chapitre, nous nous intéressons aux ***communautés par percolation de cliques ou communautés CPM (Clique Percolation Method)***, qui ont la spécificité de se chevaucher. Dans le contexte des réseaux sociaux par exemple, chaque personne appartient à plusieurs communautés telles que les collègues, la famille, les activités de loisirs, *etc.*

Bien que la correspondance entre les communautés CPM et celles observées dans le monde réel soit difficile à caractériser de manière générale, les avantages de cette définition sont bien connus [GLM13] : elle est formellement bien définie, totalement déterministe, n'utilise aucune heuristique ou optimisation de fonction difficile à interpréter, permet aux communautés de se chevaucher et chaque communauté est définie localement.

Jusqu'à présent, l'algorithme CPM n'a pas pu être étendu aux grands graphes pour des petites valeurs de k , comprises entre 5 et 10. Nous cherchons donc dans ce travail à étendre le calcul de CPM à des graphes de plus grande taille. Un obstacle pour la plupart des contributions de l'état de l'art est la mémoire requise par l'algorithme. En effet, les méthodes exactes doivent stocker en mémoire soit toutes les $(k - 1)$ -cliques, soit toutes les cliques maximales, ce qui est prohibitif dans le cas des graphes les plus massifs sur lesquels nous travaillons.

Notre contribution dans ce chapitre est double :

1. Nous améliorons l'état de l'art de la détection des communautés CPM, en tirant parti d'un algorithme existant capable d'énumérer l'ensemble des k -cliques de manière très efficace [DBS18].
2. Dans les cas où ce premier algorithme est confronté à des limitations de mémoire, nous proposons un nouvel algorithme, CPMZ, qui fournit une solution proche de la solution de CPM, en utilisant plus de temps, mais moins de mémoire. Cet algorithme s'appuie sur une nouvelle structure de donnée, qui ***généralise l'Union-Find à des ensembles non disjoints***.

Nous réalisons une étude expérimentale de nos implémentations de CPM et de CPMZ. D'une part, notre implémentation de CPM permet de calculer des solutions exactes dans des cas où cela n'était pas possible auparavant avec l'état de l'art. D'autre part, notre algorithme CPMZ permet de calculer des communautés proches

du résultat exact dans les cas où le graphe est si massif qu'aucune implémentation de CPM ne termine en raison des limitations de mémoire.

Dans tout ce chapitre, lorsque ce n'est pas précisé, $G = (V, E)$ est un graphe, $k \geq 3$ et z tel que $2 \leq z < k - 1$. C_k désigne toujours une k -clique, C_{k-1} une $(k-1)$ -clique et C_z une z -clique.

Le reste de ce chapitre est organisé comme suit. Dans la section 3.2, nous présentons en détail l'algorithme CPM qui est le point de départ de notre travail, à la fois dans ce chapitre et dans le chapitre 6 qui aborde la détection de communautés dans les réseaux temporels. Puis, dans la section 3.3, nous présentons l'état de l'art général autour de cette détection de communautés. Dans la section 3.4, nous introduisons l'idée principale de notre algorithme CPMZ, qui consiste à enregistrer les z -cliques à la place des $(k-1)$ -cliques, ce qui est moins coûteux en mémoire pour les graphes massifs du monde réel, et nous montrons dans quelle mesure cela génère des imprécisions pour la détection de communautés par rapport à l'algorithme CPM. Dans la section 3.5, nous présentons la structure de données ***Overlapping Union-Find***, qui étend l'Union-Find à des ensembles non disjoints, ce qui fournit la base de notre algorithme CPMZ, que nous décrivons dans la section 3.6. Puis, nous analysons cet algorithme dans la section 3.7, et notamment ses besoins théoriques en matière de temps et de mémoire. Enfin, dans la section 3.8, nous décrivons et évaluons les performances de notre implémentation de CPM par rapport à l'état de l'art, puis nous comparons les résultats et les performances de nos deux implémentations de CPM et de CPMZ. Nous concluons par la section 3.9, avec quelques perspectives pour les futurs travaux sur ce sujet.

3.2 Définition de CPM et algorithme

La méthode de détection de communautés par percolation de k -cliques produit ce que l'on appelle ici des communautés CPM, ou communautés de k -cliques. Une communauté CPM est un ensemble de sommets provenant d'un ensemble de k -cliques du graphe regroupées par une relation d'adjacence. Nous en donnons une définition formelle ci-dessous.

Définition 3.1 (k -cliques adjacentes). *Deux k -cliques C_k et C'_k sont dites **adjacentes** lorsqu'elles partagent $k-1$ nœuds, c'est-à-dire telles que $|C_k \cap C'_k| = k-1$.*

Définition 3.2 (Communauté CPM). *Une **communauté CPM** (ou **communauté de k -cliques**) d'un graphe désigne l'ensemble des sommets qui appartiennent à un ensemble maximal de k -cliques pouvant être reliées deux à deux par une série de k -cliques adjacentes.*

La figure 3.1 donne un exemple d'un graphe et de ses deux communautés CPM, obtenues en regroupant ses 4-cliques par la relation d'adjacence.

Notez que si un noeud n'appartient pas à au moins une k -clique, il n'appartient à aucune communauté. Néanmoins, il existe des méthodes pour étendre les communautés à ces sommets. C'est le cas par exemple de la propagation d'étiquettes, qui est une méthode itérative où à chaque étape, chaque noeud adopte l'étiquette que la plupart de ses voisins possède [RAK07].

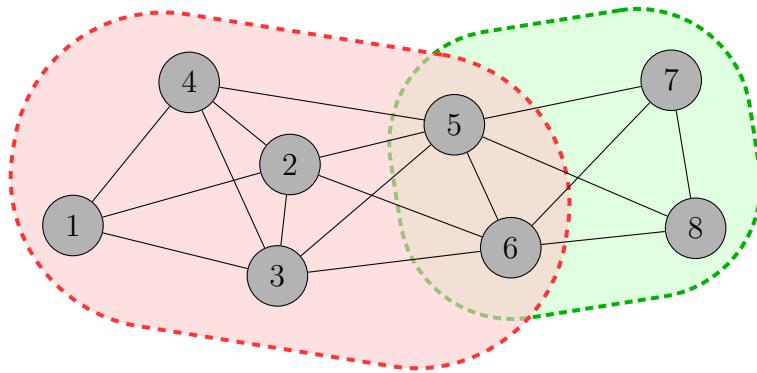


FIGURE 3.1 – Exemple d'un graphe et ses deux communautés CPM calculées pour $k = 4$. Le graphe contient quatre 4-cliques : $\{1, 2, 3, 4\}$, $\{2, 3, 4, 5\}$, $\{2, 3, 5, 6\}$ et $\{5, 6, 7, 8\}$. La clique $\{1, 2, 3, 4\}$ est adjacente à $\{2, 3, 4, 5\}$, elle-même adjacente à $\{2, 3, 5, 6\}$, ce qui forme la communauté en rouge. La clique $\{5, 6, 7, 8\}$ n'est adjacente à aucune autre, donc elle forme sa propre communauté, en vert.

Dans cette thèse, l'algorithme de départ que nous utilisons pour la détection de communautés CPM dans les graphes est l'algorithme introduit par Kum-pula *et al.* [Kum+08], qui consiste à identifier une communauté à l'ensemble des $(k - 1)$ -cliques appartenant aux k -cliques qui la composent : les communautés sont vues comme des ensembles disjoints de $(k - 1)$ -cliques. Nous verrons dans la section 3.2.2 le détail de la mise en œuvre de cet algorithme, après avoir présenté la structure de données ***Union-Find*** dans la section 3.2.1, également connue dans la littérature sous le nom de ***disjoint-set data structure***. C'est une structure qui stocke une collection d'ensembles disjoints, permettant des opérations d'union et de recherche d'éléments très efficaces entre eux.

3.2.1 Structure de données Union-Find

L'Union-Find est une structure de données qui permet de travailler sur des ensembles ***disjoints*** [Tar79]. Elle offre des opérations très efficaces pour faire l'union d'ensembles entre eux (**Union**), ainsi que pour rechercher à quel ensemble appartient un élément (**Find**). En revanche, elle ne représente pas directement les ensembles. Elle est par exemple peu efficace pour parcourir tous les éléments contenus dans un ensemble donné. Cette structure est donc très adaptée lorsqu'on a besoin de mani-

puler des ensembles sans avoir à les parcourir, ce qui est le cas dans l’algorithme CPM que nous présentons dans la suite.

Dans l’Union-Find, chaque ensemble correspond à un arbre et est représenté par la racine de cet arbre. Chaque élément pointe vers un noeud de l’arbre qui correspond à l’ensemble auquel il appartient. On sait alors à quel ensemble un élément appartient en remontant de parent en parent à la racine. Le tableau associatif UF.id permet de connaître le noeud vers lequel pointe chaque élément : $\text{UF.id}(x)$ contient le noeud associé à x dans l’Union-Find UF . L’élément x appartient alors à l’ensemble identifié par la racine de l’arbre contenant ce noeud.

La figure 3.2 (a) donne un exemple de structure Union-Find, qui représente deux ensembles disjoints. Chaque ensemble correspond à un arbre de l’Union-Find (à gauche), et chaque élément pointe vers un noeud de l’Union-Find (à droite). Par exemple, l’élément e_5 pointe vers le noeud e , ce qui signifie que e_5 est dans le même ensemble que tous les éléments dont le noeud se trouve dans le même arbre que e . Cela correspond donc à l’ensemble $\{e_4, e_5, e_6, e_7\}$. Si on veut connaître l’identifiant de l’ensemble auquel il appartient, il suffit de remonter de parent en parent jusqu’à la racine de l’arbre. Ainsi, l’identifiant de l’ensemble auquel appartient e_5 est c .

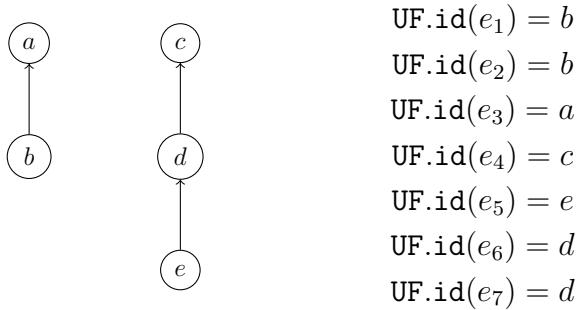
Opérations

Les opérations efficaces d’une structure d’Union-Find UF sont implémentées de la manière suivante et illustrées dans la figure 3.2 :

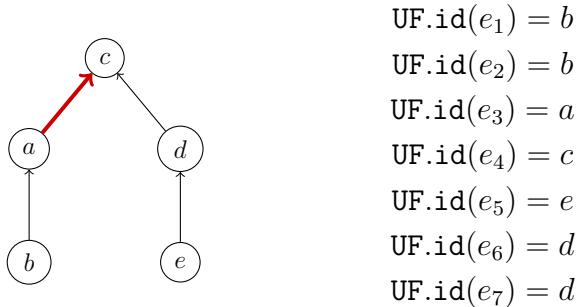
$\text{UF.Union}(p, q)$: cette fonction réalise l’union de deux ensembles dont les racines de leur arbre associé sont p et q . Pour cela, elle attache les racines p et q entre elles. Ainsi, p devient l’enfant de q ou q l’enfant de p . L’arbre résultant correspond à l’union des deux ensembles, qui est représentée par la nouvelle racine (p ou q). La figure 3.2 (b) montre le résultat de l’union réalisée sur les deux ensembles disjoints de la structure UF de la figure 3.2 (a).

$\text{UF.Find}(x)$: cette fonction renvoie le représentant p de l’ensemble auquel l’élément x appartient. C’est-à-dire qu’elle renvoie la racine de l’arbre contenant le noeud auquel l’élément x est associé. Pour cela, elle part du noeud de x et elle remonte l’arbre jusqu’à arriver à sa racine p . Par exemple, dans la figure 3.2 (a), un Find sur e_1 part du noeud b , remonte jusqu’à la racine a , et indique que e_1 appartient à l’ensemble dont l’identifiant est a .

$\text{UF.MakeSet}(x)$: cette fonction crée un nouvel ensemble $\{x\}$, en créant un nouvel arbre contenant un sommet q et en associant x à q par la création d’une nouvelle entrée $\text{UF.id}(x) = q$. Elle renvoie q .



(a) État d'une structure Union-Find UF représentant deux ensembles disjoints $\{e_1, e_2, e_3\}$ et $\{e_4, e_5, e_6, e_7\}$. Les arbres de l'Union-Find sont représentés à gauche, il y en a un pour chaque ensemble. Chaque élément e_i pointe vers un nœud de ces arbres, et appartient à l'ensemble correspondant. Les identifiants des nœuds vers lesquels pointe chaque e_i sont affichées à droite par le tableau associatif UF.id.



(b) Structure UF de la figure (a) après avoir réalisé l'opération `UF.Union(a, c)`. Cette opération fait l'union des deux ensembles disjoints représentés par les racines a et c , en ajoutant simplement une arête de a vers c . La structure représente alors l'ensemble $\{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$. Notez que c'est l'arbre le plus petit qui a été attaché à l'arbre le plus grand, afin de ne pas agrandir la hauteur de l'arbre résultant.

FIGURE 3.2 – Exemple de manipulation d'une structure Union-Find UF. La figure (a) représente deux ensembles disjoints qui sont fusionnés dans la figure (b) par une opération d'Union.

Procédures algorithmiques

Pour que les opérations **Union** et **Find** soient les plus efficaces possibles, deux procédures algorithmiques sont à mettre en place lors de leur utilisation :

Union(p,q) (union par rang) : Lorsqu'est effectuée l'union de deux ensembles représentés par les arbres dont les racines sont p et q , nous comparons les hauteurs respectives de ces arbres. La racine de l'arbre le moins haut devient le fils de la racine de l'arbre le plus haut. Ainsi, la hauteur de l'arbre résultant de l'union ne s'accroît que si les deux arbres ont la même taille. Par exemple, dans la figure 3.2 (b), c'est la racine a que l'on connecte à c , et non l'inverse.

Find(x) (compression de chemin) : L'opération **Find** part d'un nœud d'un arbre et remonte à sa racine. Elle peut être améliorée dans le but de réduire la taille des chemins des noeuds à leur racine. Pour cela, chaque nœud parcouru lors d'une opération **Find** est directement redirigé vers la racine de son arbre. De plus, après un appel à **Find(x)**, nous changeons le nœud vers lequel pointe x , en le faisant pointer vers la racine de son arbre. Ainsi, lors des prochains appels impliquant x ou l'un des nœuds redirigés, le chemin parcouru dans l'arbre sera plus court. Ces deux opérations sont illustrées par la figure 3.3, où l'appel **Find(e_2)** est réalisé : le nœud associé à e_2 est changé en la racine c , et on fait pointer le nœud b , par lequel la recherche est passée, directement vers la racine c .

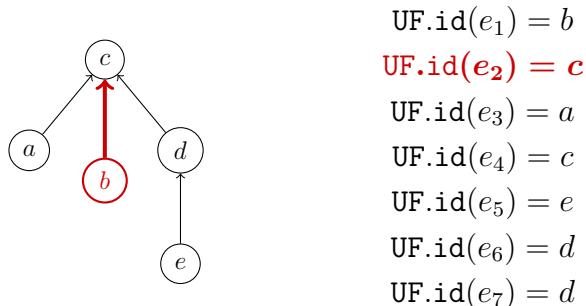


FIGURE 3.3 – Structure UF de la figure 3.2 (b) après avoir réalisé l'opération **UF.Find(e_2)**. Cette fonction renvoie l'identifiant de l'ensemble auquel appartient e_2 , c'est-à-dire la racine c de l'arbre auquel il appartient. Deux modifications internes sont réalisées au cours de l'exécution : le nœud de e_2 est remplacé par la racine c , et le chemin de b à la racine c est raccourci. Ainsi, les prochains appels à **Find** faisant intervenir e_2 ou b (par exemple **Find(e_1)**) seront plus efficaces.

Complexités associées

Les opérations décrites ci-dessus sont extrêmement efficaces. Elles se font toutes en temps constant, à l'exception de l'opération **Find** qui nécessite de remonter un

arbre de l’Union-Find jusqu’à la racine. Cette complexité peut donc être en théorie en $\mathcal{O}(n)$, où n est le nombre d’éléments dans l’Union-Find. Néanmoins, il est démontré grâce aux implémentations efficaces que nous venons de décrire que sa complexité est en fait en $\mathcal{O}(\alpha(n))$, où α est l’inverse de la fonction d’Ackermann [Tar79]. En théorie, cette fonction croît vers l’infini, mais très lentement, et en pratique elle ne dépasse pas 5. En effet, il s’agit d’une fonction croissante et telle que $\alpha(10^{35\,000}) \leq 5^1$. Ainsi, nous pouvons considérer que cette fonction $\alpha(n)$ ne dépasse jamais 5 en pratique, et qu’elle se trouve dans $\mathcal{O}(1)$.

Pour les énoncés de complexité des algorithmes qui utilisent la structure Union-Find, nous supposons que les améliorations détaillées ci-dessus sont effectuées. Ainsi, dans toute la suite de la thèse, chaque appel à `Find` dans une structure d’Union-Find sera considéré en $\mathcal{O}(1)$.

3.2.2 Algorithme CPM

Principe général

L’algorithme de résolution du problème CPM est basé sur l’énumération à la volée des k -cliques du graphe d’entrée, afin de mettre à jour une structure de communautés qui représente les cliques déjà traitées, et qui correspond à l’ensemble des communautés CPM à la fin de l’énumération. Il repose sur le fait que les communautés CPM peuvent être identifiées à un partitionnement des k -cliques, de telle sorte que deux k -cliques sont dans une même partition lorsqu’elles partagent $(k - 1)$ nœuds. Une communauté CPM peut donc être vue comme une composante connexe du graphe dont les nœuds sont les k -cliques connectées par des arêtes lorsqu’elles sont adjacentes. Par exemple, dans la figure 3.1, les trois cliques $\{1, 2, 3, 4\}$, $\{2, 3, 4, 5\}$ et $\{2, 3, 5, 6\}$ forment une composante connexe du graphe que nous venons de décrire.

Ainsi, pour calculer l’ensemble des communautés, il suffit de traiter une à une chaque k -clique, à la volée, en l’ajoutant à une communauté qui contient une k -clique qui lui est adjacente. S’il y en a plusieurs, elles sont alors fusionnées entre elles, et s’il n’y en a aucune, une nouvelle est créée. À la fin de l’énumération, l’ensemble des communautés obtenu forme les communautés CPM.

Pseudo-code de l’algorithme

Tout d’abord, nous nous appuyons sur une idée introduite par Kumpula *et al.* [Kum+08]. Une communauté CPM peut être identifiée à l’ensemble des $(k - 1)$ -cliques des k -cliques qui la composent. Ainsi, lors du traitement d’une k -clique, les communautés auxquelles on doit l’ajouter sont celles qui contiennent l’une de ses $(k - 1)$ -cliques : elles sont fusionnées s’il y en a plusieurs, et une nouvelle est créée s’il n’y en a aucune. De plus, une $(k - 1)$ -clique ne peut pas appartenir

1. <https://www.gabrielnivasch.org/fun/inverse-ackermann>

à deux communautés différentes, car sinon cela signifie qu'une k -clique contenant cette $(k - 1)$ -clique a été ajoutée à chacune de ces deux communautés, ce qui est impossible, car cela impliquerait qu'elles aient été fusionnées.

Ces communautés peuvent donc être représentées par une structure Union-Find représentant des ***ensembles disjoints de $(k - 1)$ -cliques***. L'algorithme énumère toutes les k -cliques du graphe G , et il construit les communautés CPM au fur et à mesure de l'énumération. Le pseudo-code de cette procédure est donné par l'algorithme 3.1, et il est illustré par la figure 3.4.

Algorithme 3.1: Algorithme CPM.

Entrée: Graphe $G = (V, E)$; $k \in \llbracket 3, +\infty \rrbracket$.

Sortie: Structure Union-Find représentant les communautés CPM de G .

```

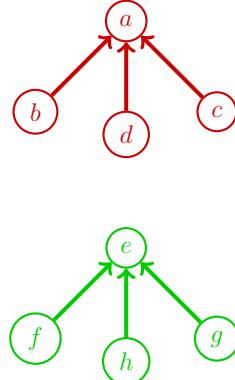
1 UF ← structure Union-Find vide
2 for each  $k$ -clique  $C_k$  de  $G$  do
3    $p \leftarrow -1$ 
4   for each  $u \in C_k$  do
5      $C_{k-1} \leftarrow C_k \setminus \{u\}$ 
6     if  $C_{k-1} \in \text{UF}$  then
7        $q \leftarrow \text{UF.Find}(C_{k-1})$ 
8     else
9        $q \leftarrow \text{UF.MakeSet}(C_{k-1})$ 
10     $p \leftarrow \text{UF.Union}(p, q)$ 

```

L'algorithme 3.1 traite chaque k -clique du graphe une par une (ligne 2). Pour chaque k -clique C_k , il itère sur ses $(k - 1)$ -cliques C_{k-1} (ligne 5). Pour chacune des C_{k-1} , il regarde la communauté à laquelle elle appartient dans l'Union-Find (ligne 7). Si elle n'appartient encore à aucune communauté, une nouvelle entrée est créée pour C_{k-1} (ligne 9). Puis, à la ligne 10, il réalise l'union de cette communauté avec celle des autres $(k - 1)$ -cliques. Notez que pour la première $(k - 1)$ -clique traitée, l'union de -1 et de q renvoie q . Ainsi, à la fin du traitement de C_k , chacune de ses $(k - 1)$ -cliques appartient à la même communauté. Cette communauté correspond à la fusion des communautés de chacune des $(k - 1)$ -cliques de C_k avant son traitement, ou bien à une nouvelle si aucune de ses $(k - 1)$ -clique n'avait encore été assignée à une communauté.

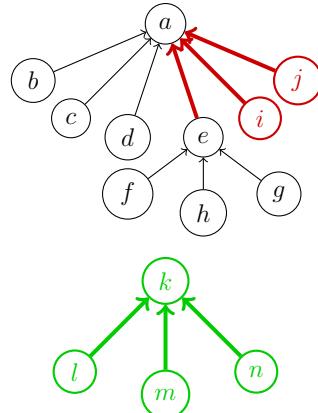
Enfin, pour obtenir l'ensemble des sommets des communautés calculées, il est nécessaire de réaliser un post-traitement. Il consiste à parcourir l'Union-Find UF en ajoutant les sommets de chaque $(k - 1)$ -clique à la communauté à laquelle elle appartient.

UF.id(1, 2, 3) = *a*
 UF.id(1, 2, 4) = *b*
 UF.id(1, 3, 4) = *c*
 UF.id(2, 3, 4) = *d*
 UF.id(2, 3, 5) = *e*
 UF.id(2, 3, 6) = *f*
 UF.id(2, 5, 6) = *g*
 UF.id(3, 5, 6) = *h*



(a) Après le traitement de la première k -clique $\{1, 2, 3, 4\}$, chacune de ses $(k-1)$ -clique pointe vers un nœud différent de l'UF, du fait de la ligne 9. La ligne 10 connecte chacun de ces nœuds en formant l'arbre à droite en rouge, qui signifie que ces $(k-1)$ -cliques appartiennent à la même communauté, identifiée par le sommet *a*. Il en est de même pour la deuxième k -clique traitée, $\{2, 3, 5, 6\}$, qui forme l'arbre de droite en vert.

UF.id(1, 2, 3) = *a*
 UF.id(1, 2, 4) = *b* UF.id(2, 4, 5) = *i*
 UF.id(1, 3, 4) = *c* UF.id(3, 4, 5) = *j*
UF.id(2, 3, 4) = *a* UF.id(5, 6, 7) = *k*
UF.id(2, 3, 5) = *a* UF.id(5, 7, 8) = *l*
 UF.id(2, 3, 6) = *f* UF.id(5, 6, 8) = *m*
 UF.id(2, 5, 6) = *g* UF.id(6, 7, 8) = *n*
 UF.id(3, 5, 6) = *h*



(b) Au moment où la k -clique $\{2, 3, 4, 5\}$ est traitée, sa $(k-1)$ -clique $\{2, 3, 4\}$ est dans la communauté identifiée par *a*, et sa $(k-1)$ -clique $\{2, 3, 5\}$ est dans la communauté identifiée par *e*. Ces deux communautés sont donc fusionnées par l'Union de la ligne 10, et deux nœuds *i* et *j* sont ajoutés à cet arbre, correspondant aux deux autres $(k-1)$ -cliques qui n'ont encore jamais été vue, $\{2, 4, 5\}$ et $\{3, 4, 5\}$. Enfin, la dernière k -clique $\{5, 6, 7, 8\}$ crée sa propre communauté, car elle n'a aucune $(k-1)$ -clique en commun avec la communauté identifiée par *a*.

FIGURE 3.4 – Exemple de déroulé de l'algorithme 3.1 CPM, avec $k = 4$, sur le graphe de la figure 3.1, qui contient quatre 4-cliques et deux communautés CPM. La figure (a) représente la structure UF après le traitement de deux k -cliques du graphe, et la figure (b) après le traitement des deux restantes, qui représente donc ses communautés CPM.

3.2.3 Complexité de l'algorithme CPM

Pour chaque k -clique de G , l'algorithme 3.1 CPM effectue un appel à `Union` et à `Find` ou `MakeSet` pour chacune de ses $(k - 1)$ -cliques, et on a vu que ces opérations étaient en $\mathcal{O}(1)$. Néanmoins, un accès à la valeur associée à C_{k-1} dans le tableau associatif UF se fait en $\mathcal{O}(k - 1)$. Par conséquent, la complexité de l'algorithme 3.1 est en $\mathcal{O}(k^2 \cdot n_k + c(k))$, où n_k désigne le nombre de k -cliques du graphe, et $c(k)$ la complexité de l'énumération des k -cliques. Notez que le nombre de nœuds dans l'Union-Find est au plus de $k \cdot n_k$ et donc que le post traitement s'exécute lui aussi en $\mathcal{O}(k^2 \cdot n_k)$.

D'après le théorème 5.7 de Danisch *et al.* [DBS18], nous savons que n_k est en $\mathcal{O}\left(m \cdot \left(\frac{c}{2}\right)^{k-2}\right)$, où c est la dégénérescence du graphe. De plus, Danisch *et al.* montrent que la complexité de l'énumération des k -cliques est en $\mathcal{O}\left(k \cdot m \cdot \left(\frac{c}{2}\right)^{k-2}\right)$. Ainsi, la complexité temporelle de l'algorithme CPM est en $\mathcal{O}\left(k^2 \cdot m \cdot \left(\frac{c}{2}\right)^{k-2}\right)$.

La complexité spatiale de cet algorithme est dominée par la taille des arbres de l'Union-Find. Cette structure contient un nœud pour chaque $(k - 1)$ -clique de G qui est incluse dans une k -clique. Ainsi, la complexité en mémoire de l'algorithme est en $\mathcal{O}((k - 1) \cdot n_{k-1})$, qui est donc en $\mathcal{O}\left((k - 1) \cdot m \cdot \left(\frac{c}{2}\right)^{k-3}\right)$.

On voit ainsi que la dégénérescence du graphe joue un rôle important dans ces complexités, ainsi que le facteur k . Ceci explique pourquoi dans nos expériences, l'algorithme fonctionne bien sur les graphes du monde réel, dont la dégénérescence est faible, pour de petites valeurs de k .

3.3 État de l'art

Les communautés CPM ont été définies pour la première fois en 2005 par Palla *et al.* [Pal+05]. Depuis, plusieurs algorithmes ont été développés pour améliorer l'efficacité de cette détection de communautés. Dans l'état de l'art, les algorithmes qui existent pour calculer ces communautés peuvent être divisés en deux catégories :

- (1) La plupart des approches énumèrent l'ensemble des ***cliques maximales*** de taille k ou plus et les utilisent pour calculer l'ensemble des communautés CPM [GLM13; Pal+05; RMH12]. En effet, deux cliques maximales qui se chevauchent sur $k - 1$ nœuds ou plus peuvent être jointes, leurs sommets appartiennent à la même communauté de k -cliques².

2. Il est évident que les sommets d'une clique maximale sont tous dans une même communauté. Si deux clique maximales se chevauchent sur $k - 1$ sommets, alors en prenant un sommet en plus dans chaque clique maximale, cela fournit deux k -cliques adjacentes, qui connectent donc ces deux

- (2) Les travaux réalisés par Kumpula *et al.* [Kum+08] dont nous avons présenté la méthode dans la section 3.2 précédente, partent quant à eux de l'énumération de l'ensemble des **k -cliques** du graphe. Comme on l'a vu, ils calculent les communautés CPM en suivant strictement la définition, c'est-à-dire en regroupant entre elles les k -cliques qui sont adjacentes.

On notera que si le graphe d'entrée comporte une grande clique, par exemple une clique de taille 1000, et que l'on s'intéresse aux communautés de 10-cliques, l'algorithme de la catégorie (1) semble alors plus intéressant. En effet, il y a au moins $\binom{1000}{10} > 10^{23}$ 10-cliques dans le graphe d'entrée, ce qui est prohibitif. De plus, cette clique de 1000 est en fait incluse dans une seule communauté de 10-cliques et peut être trouvée directement et efficacement à l'aide d'un algorithme qui énumère les cliques maximales. C'est la raison principale pour laquelle il y a plus de méthodes qui suivent l'approche de l'énumération des cliques maximales (catégorie (1)) plutôt que l'énumération des k -cliques (catégorie (2)).

Toutefois, il a été constaté que la plupart des graphes du monde réel ne contiennent pas de très grandes cliques et que l'énumération des k -cliques pour des petites valeurs de k est un problème qui se résout rapidement en pratique sur ces graphes [DBS18 ; Li+20]. Cela rend les algorithmes de la catégorie (2) plus intéressants pour les scénarios pratiques, contrairement à ce qui était pensé au départ.

L'algorithme détaillé dans [Kum+08] est le seul algorithme de la catégorie (2). Il s'agit de l'algorithme 3.1. Nous l'avons implémenté en utilisant un meilleur algorithme pour énumérer l'ensemble des k -cliques du graphe [DBS18], ce qui rend l'implémentation plus rapide que la leur. L'algorithme CPMZ que nous présentons dans la suite de ce chapitre relève lui aussi de la catégorie (2). Il s'agit d'une extension directe de cet algorithme CPM, qui permet d'utiliser moins de mémoire, et donc de passer le calcul des communautés à l'échelle de graphes plus massifs.

3.4 CPMZ : représenter les communautés comme des ensembles de z -cliques, $1 \leq z < k - 1$

Dans cette section, nous présentons le cadre qui permet de développer l'algorithme CPMZ. Pour une valeur de k donnée, nous considérons des entiers z tels que $1 \leq z < k - 1$. Nous souhaitons utiliser les z -cliques pour calculer une version des communautés CPM, mais cela ne marche pas parfaitement pour avoir les communautés exactes. Nous discutons cela dans cette section, avant de présenter les détails algorithmiques dans les sections 3.5 et 3.6.

Dans la section 3.4.1, nous justifions qu'il est plus efficace en mémoire de stocker

cliques maximales en une seule communauté.

les l'ensemble des z -cliques plutôt que celui des $(k - 1)$ -cliques ce qui justifie la volonté de les utiliser pour calculer les communautés ; puis nous introduisons dans la section 3.4.2 la notion de $(k - 1)$ -clique parasite d'un ensemble de z -cliques qui est la cause du fait qu'on ne puisse pas obtenir les communautés CPM exactes en utilisant les z -cliques ; ce qui nous conduit à définir dans la section 3.4.3 les communautés CPMZ que nous calculons par la suite.

3.4.1 Stocker les z -cliques plutôt que les $(k - 1)$ -cliques

On a vu que l'algorithme 3.1 CPM crée un nœud dans l'Union-Find pour chaque $(k - 1)$ -clique qui est incluse dans une k -clique. Or, lorsque k augmente, le nombre de $(k - 1)$ -cliques peut devenir très grand, ce qui rend cette approche problématique pour l'étude des graphes massifs du monde réel, du point de vue de la mémoire que cela nécessite.

Pour les graphes où il n'est pas possible de stocker toutes les $(k - 1)$ -cliques en mémoire, il peut rester possible de stocker l'ensemble des z -cliques. Pour des valeurs de z relativement petites par rapport à k , il y a beaucoup moins de z -cliques que de $(k - 1)$ -cliques dans les graphes. Pour comprendre ce phénomène, considérons le cas d'une grande clique de taille c . Elle contient $\binom{c}{z}$ z -cliques et ce nombre augmente avec z pour $z < c/2$. Ainsi, par exemple, si on prend $c = 20$, cette clique contient environ 200 2-cliques (arêtes), 5 000 4-cliques et 40 000 6-cliques.

Nous utilisons cette idée pour proposer une nouvelle manière de considérer les communautés CPM, moins coûteuses en mémoire. Au lieu de considérer une communauté comme l'ensemble des $(k - 1)$ -cliques des k -cliques qui la composent, nous proposons ici de l'enregistrer comme l'ensemble des z -cliques de ses k -cliques. Alors, nous faisons la simplification suivante : une $(k - 1)$ -clique appartient à cette communauté lorsque l'ensemble de ses z -cliques y sont. Il s'agit bien évidemment d'une condition nécessaire, car une communauté qui contient une $(k - 1)$ -clique contient toujours ses z -cliques. Mais ce n'est pas une condition suffisante, comme nous le montrons dans la section suivante.

3.4.2 $(k - 1)$ -cliques parasites d'un ensemble de z -cliques

Si on considère qu'une communauté de k -cliques est stockée comme l'ensemble de ses z -cliques, alors on introduit une perte d'information : une $(k - 1)$ -clique ayant toutes ses z -cliques dans la communauté peut n'appartenir à aucune k -clique de la communauté. Nous appelons une telle $(k - 1)$ -clique une clique parasite de cette communauté. Nous en donnons une définition ci-dessous et une illustration dans la figure 3.5.

Définition 3.3. Une $(k - 1)$ -clique est appelée **$(k - 1)$ -clique parasite** d'une

communauté de k -cliques lorsqu'elle n'est incluse dans aucune k -clique de la communauté, alors que chacune de ses z -cliques le sont.

La figure 3.5 (a) contient une $(k - 1)$ -clique parasite, pour $k = 4$: la $(k - 1)$ -clique centrale avec les noeuds $\{4, 6, 7\}$, qui est formée par les z -cliques (arêtes) d'autres k -cliques. Elle ne fait elle-même partie d'aucune k -clique de la communauté. Ainsi, elle est considérée comme un point d'attache à la communauté alors qu'elle ne l'est pas pour l'algorithme CPM.

Le problème de l'existence des cliques parasites est que, dans l'algorithme CPM, si une $(k - 1)$ -clique apparaît dans une communauté alors qu'elle ne devrait pas, alors elle peut faire fusionner cette communauté avec d'autres qui contiennent effectivement cette $(k - 1)$ -clique. C'est le cas dans l'exemple de la figure 3.5 (b) : la k -clique $\{4, 6, 7, 10\}$ contient la $(k - 1)$ -clique parasite $\{4, 6, 7\}$. Elle est donc identifiée comme faisant partie de cette communauté, alors que ce n'est pas le cas pour l'algorithme CPM.

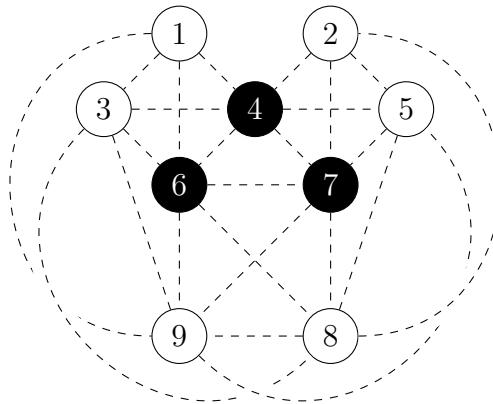
Remarquons que plus z est élevé, c'est-à-dire plus z est proche de $k - 1$, et moins il y a de $(k - 1)$ -cliques parasites (et si on autorise $z = k - 1$, il n'y en a aucune, mais cela revient à la méthode CPM). En effet, si $z_2 > z_1$, une clique parasite pour z_2 l'est aussi pour z_1 , car les z_1 -cliques sont incluses dans les z_2 -cliques ; mais la réciproque n'est pas vraie. Ainsi, faire augmenter z diminue les imprécisions, mais augmente le nombre de z -cliques et donc la mémoire (dans les graphes massifs que nous étudions).

L'hypothèse que nous faisons est qu'une communauté qui contient toutes les z -cliques d'une $(k - 1)$ -clique a de fortes chances de contenir cette $(k - 1)$ -clique. Cette hypothèse sera validée par notre étude expérimentale, qui montre que le résultat qu'on obtient par la suite en introduisant cette approximation est en fait très proche de celui de l'algorithme CPM (voir section 3.8.3).

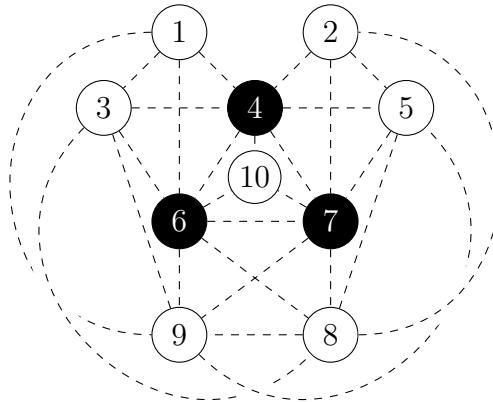
3.4.3 Définition des communautés CPMZ

Dans l'algorithme 3.1 CPM, une communauté est représentée comme un ensemble de $(k - 1)$ -cliques et les communautés CPM correspondent à des ensembles ***disjoints*** de $(k - 1)$ -cliques. Dans ce qui suit, une communauté CPMZ est représentée comme un ensemble de z -cliques, et les communautés CPMZ sont représentées comme des ensembles ***non disjoints*** de z -cliques.

Le problème que nous traitons alors consiste à calculer ce que l'on appelle des communautés CPMZ. Elles sont définies formellement ci-dessous, comme étant une



(a) Dans cet exemple, $k = 4$ et $z = 2$. La communauté représentée correspond à l'ensemble des z -cliques (arêtes) de ses k -cliques. La $(k - 1)$ -clique du milieu $\{4, 6, 7\}$ est une $(k - 1)$ -clique parasite de cette communauté vue comme un ensemble de z -cliques.



(b) La k -clique $\{4, 6, 7, 10\}$ contient la $(k - 1)$ -clique parasite $\{4, 6, 7\}$, elle est donc perçue comme devant être rattachée à cette communauté vue comme un ensemble de z -cliques, alors qu'elle n'est adjacente à aucune de ses k -cliques.

FIGURE 3.5 – Dans cet exemple, $k = 4$ et $z = 2$. La figure (a) représente une communauté de k -cliques formée par les nœuds des k -cliques suivantes : $\{1, 3, 4, 6\}$, $\{1, 3, 6, 9\}$, $\{3, 6, 8, 9\}$, $\{6, 7, 8, 9\}$, $\{5, 7, 8, 9\}$, $\{2, 5, 7, 8\}$, $\{2, 4, 5, 7\}$. Dans la figure (b), une k -clique supplémentaire est prise en compte : $\{4, 6, 7, 10\}$.

agrégation des communautés CPM du graphe.

Définition 3.4 (Communautés CPMZ). *Un ensemble de communautés CPMZ d'un graphe G est un ensemble de communautés agglomérées, correspondant aux communautés CPM de G dont certaines ont été fusionnées entre elles. On appelle communauté CPMZ une communauté issue d'un tel ensemble.*

Notre méthode efficace en mémoire, appelée **algorithme CPM**, étant donnés un graphe G , la taille k des k -cliques et un entier $z \in [2, k - 2]$, renvoie un ensemble de communautés agglomérées de k -cliques, de sorte que chaque communauté CPM est incluse dans une et une seule communauté CPMZ (voir le théorème 3.1 dans la section 3.7). Nous détaillons l'algorithme dans la section 3.6, après avoir introduit la structure d'Overlapping Union-Find sur laquelle il repose dans la section 3.5 suivante.

3.5 Overlapping Union-Find : adapter l'Union-Find à des ensembles non disjoints

Pour réaliser des unions d'ensemble disjoints et chercher à quel ensemble appartient un élément de manière très efficace, nous avons vu dans la section 3.2.1 que l'Union-Find était une structure très adaptée.

Néanmoins, nous souhaitons à présent réaliser des unions d'ensembles non disjoints (de z -cliques). Pour cela, nous développons une version adaptée de l'Union-Find, que l'on appelle **Overlapping Union-Find (OUF)**.

Par rapport à l'Union-Find, sa définition est telle qu'un élément peut appartenir à plusieurs ensembles, et donc être associé à des noeuds de plusieurs arbres. Elle associe à chaque élément x , l'ensemble $\text{OUF.id}(x)$ des nœuds des arbres qui correspondent aux ensembles contenant x . Alors, à partir de chaque nœud associé à x , nous pouvons remonter à la racine de l'arbre de ce nœud, qui caractérise l'un des ensembles auquel il appartient.

Opérations

Une instance d'Overlapping Union-Find OUF donnée implémente les fonctions suivantes, qui sont illustrées par la figure 3.6 :

- $\text{OUF.Union}(\{p_1, \dots, p_r\})$: réalise l'union des ensembles dont les racines de leur arbre associé sont p_1, \dots, p_r . Pour cela, elle choisit une racine p_i , et y attache les autres racines. L'arbre résultant correspond à l'union de ces r ensembles, qui est représentée par la racine p_i qui a été choisie. Il s'agit de la même opération que dans le cas de l'Union-Find, qu'on autorise sur plusieurs arbres en même temps afin de simplifier l'écriture du pseudocode de notre algorithme CPMZ.

- `OUF.Find(x)` : renvoie la liste $\{q_1, \dots, q_l\}$ des représentants des ensembles auxquels x appartient. Chaque q_i est la racine d'un arbre. Lorsque $x \notin \text{OUF}$ (c'est-à-dire que x n'appartient à aucun ensemble représenté par l'`OUF`), cette opération renvoie l'ensemble vide.
- `OUF.MakeSet()` : cette fonction crée un nouvel ensemble vide, c'est-à-dire un arbre composé d'un nœud q sur lequel aucun élément ne pointe. Elle renvoie q .
- `OUF.Add(x,q)` : cette fonction ajoute l'élément x à l'ensemble représenté par q . Pour cela, elle ajoute q à l'ensemble des nœuds qui sont associés à x dans `OUF.id`.

Complexité des opérations

Nous implémentons les fonctions `OUF.Union` et `OUF.Find` avec les mêmes optimisations que celles décrites pour la structure Union-Find dans la section 3.2.1 :

- `OUF.Union` garde toujours comme parent la racine de l'arbre le plus profond afin que la profondeur de l'arbre résultant n'augmente que lorsque tous les arbres à unir sont de même hauteur.
- `OUF.Find(x)` remplace l'ensemble des nœuds de `OUF.id(x)` par ***l'ensemble des racines de leurs arbres***. Par exemple, dans la figure 3.6 (d), les noeuds associés à e_2 appartenaient tous à l'arbre dont la racine est a (voir figure 3.6 (c)), donc l'entrée `OUF.id(e_2)` devient $\{a\}$. Comme pour l'Union-Find, elle fait pointer chaque nœud parcouru directement vers sa racine.

Avec ces optimisations, de manière similaire à l'Union-Find, nous pouvons considérer que l'accès à la racine d'un nœud de l'`OUF` se fait en $\mathcal{O}(1)$. Les complexités des opérations sont alors les suivantes :

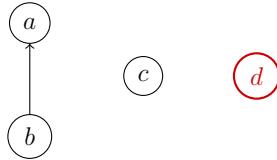
- `OUF.Union($\{p_1, \dots, p_r\}$)` : $\mathcal{O}(r)$;
- `OUF.Find(x)` : $\mathcal{O}(|\text{OUF.id}(\mathbf{x})|)$;
- `OUF.MakeSet()` : $\mathcal{O}(1)$;
- `OUF.Add(x,q)` : $\mathcal{O}(1)$.

On constate une différence notable pour la fonction `OUF.Find` par rapport à celle de l'Union-Find : sa complexité n'est plus constante. Pour pouvoir établir la complexité de la fonction, il faut connaître le nombre maximal de nœuds de l'`OUF` vers lesquels un élément peut pointer. Cela rend la structure plus coûteuse d'un point de vue théorique, mais nous verrons que dans les cas où nous l'utilisons, ce nombre est faible en pratique, ce qui rend cette complexité intéressante (voir la discussion à ce sujet dans la section 3.7, et l'étude expérimentale dans la section 3.8.2).



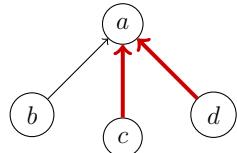
$\text{OUF.id}(e_1) = \{a\}$
 $\text{OUF.id}(e_2) = \{b, c\}$

- (a) Structure OUF représentant deux ensembles non disjoints : $\{e_1, e_2\}$ et $\{e_2\}$, identifiés respectivement par la racine de leur arbre associé, a et c .



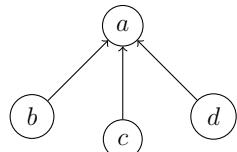
$\text{OUF.id}(e_1) = \{a, \textcolor{red}{d}\}$
 $\text{OUF.id}(e_2) = \{b, c\}$
 $\text{OUF.id}(e_3) = \{\textcolor{red}{d}\}$

- (b) Strucutre OUF représentant trois ensembles non disjoints : $\{e_1, e_2\}$, $\{e_2\}$ et $\{e_1, e_3\}$. Il s'agit de la structure de la figure (a) à laquelle l'ensemble $\{e_1, e_3\}$ a été ajouté, identifié par l'arbre à un nœud d . Pour cela, une opération OUF.MakeSet a été réalisée pour créer le nœud d , et deux opération $\text{OUF.Add}(e_1, d)$ et $\text{OUF.Add}(e_3, d)$ pour remplir cet ensemble.



$\text{OUF.id}(e_1) = \{a, d\}$
 $\text{OUF.id}(e_2) = \{b, c\}$
 $\text{OUF.id}(e_3) = \{d\}$

- (c) Structure OUF représentant l'ensemble $\{e_1, e_2, e_3\}$. Ils s'agit de la modification des arbres de l'OUF de la figure (b) après l'opération $\text{OUF.Union}(\{a, c, d\})$. Cette structure représente alors l'ensemble $\{e_1, e_2, e_3\}$, et les nœuds associés dans OUF.id restent inchangés.



$\text{OUF.id}(e_1) = \{a, d\}$
 $\text{OUF.id}(e_2) = \{\textcolor{red}{a}\}$
 $\text{OUF.id}(e_3) = \{d\}$

- (d) Structure OUF représentant l'ensemble $\{e_1, e_2, e_3\}$. Il s'agit de la modification de OUF.id de la figure (c) après l'opération $\text{OUF.Find}(e_2)$. L'ensemble de nœuds de l'OUF $\{b, c\}$ vers lequel pointait e_2 est remplacé par l'ensemble des racines des arbres de ces nœuds, à savoir $\{a\}$.

FIGURE 3.6 – Exemple d'une structure d'Overlapping Union-Find, OUF, pour laquelle on donne des exemples d'utilisation des fonctions OUF.MakeSet , OUF.Add , OUF.Union et OUF.Find .

3.6 Algorithme pour calculer les communautés CPMZ

L'idée principale de notre algorithme CPMZ est d'identifier chaque $(k - 1)$ -clique à l'ensemble des z -cliques qu'elle contient. En dehors de cette idée, l'algorithme est similaire à celui de CPM : il construit les communautés CPMZ au fur et à mesure de l'énumération des k -cliques de G . Comme avec l'algorithme CPM, pour chaque k -clique, l'objectif est de réaliser l'union des communautés identifiées à chacune de ses $(k - 1)$ -cliques (et de créer une nouvelle communauté s'il n'y en a pas). Pour cela, puisque nous considérons qu'une $(k - 1)$ -clique est représentée par l'ensemble de ses z -cliques, les communautés identifiées à chaque $(k - 1)$ -clique sont celles qui contiennent toutes ses z -cliques.

L'algorithme 3.2 donne le pseudo-code de l'algorithme CPMZ, et un exemple de son exécution est présenté dans la figure 3.7. Il utilise une structure de donnée d'Overlapping Union-Find, OUF, permettant de représenter les communautés comme des ensembles non disjoints de z -cliques. La boucle de la ligne 2 itère sur chaque k -clique C_k du graphe d'entrée G . Pour chacune de ses $(k - 1)$ -cliques C_{k-1} (ligne 5), l'intersection de la ligne 6 enregistre les communautés qui contiennent toutes ses z -cliques. Il peut y en avoir plusieurs pour C_{k-1} , mais seulement ***au plus*** une qui n'est pas une clique parasite. Néanmoins, nous ne sommes pas en capacité de détecter celles qui sont des parasites et celles qui ne le sont pas, ce qui impose de tenir compte de toutes. Cela peut entraîner une fusion erronée de deux ou plusieurs communautés, qui seraient disjointes avec CPM.

Les noeuds (communautés) à fusionner sont enregistrés dans l'ensemble S . Ils sont fusionnés par l'**Union** de la ligne 8. Néanmoins, il peut s'avérer que S soit vide. Cela correspond au cas où aucune des $(k - 1)$ -cliques de C_k n'appartient encore à une communauté. Si cela se produit, alors une nouvelle communauté est créée à la ligne 10. Notez que, contrairement à CPM, si une $(k - 1)$ -clique C_{k-1} n'a pas encore été observée dans l'algorithme, elle peut quand même être identifiée à une communauté, si cette communauté contient toutes ses z -cliques.

Enfin, l'identifiant q de la communauté résultant du traitement de C_k est ajouté à l'ensemble des identifiants des communautés auxquelles appartiennent chaque z -clique de C_k (ligne 12).

À la fin de l'algorithme, chaque arbre de l'Overlapping Union-Find correspond à une communauté CPMZ, dont les sommets qui la composent sont les sommets de ses z -cliques. Comme pour l'algorithme CPM, pour obtenir la liste des sommets de chaque communauté, il convient à la fin de cette procédure de parcourir l'Overlapping Union-Find, afin d'ajouter les sommets de chaque z -cliques aux communautés auxquelles elles appartiennent.

$\text{OUF.id}(1, 2) = \{a\}$	$\text{OUF.id}(2, 5) = \{b\}$
$\text{OUF.id}(1, 3) = \{a\}$	$\text{OUF.id}(2, 6) = \{b\}$
$\text{OUF.id}(1, 4) = \{a\}$	$\text{OUF.id}(3, 5) = \{b\}$
$\text{OUF.id}(2, 4) = \{a\}$	$\text{OUF.id}(3, 6) = \{b\}$
$\text{OUF.id}(3, 4) = \{a\}$	$\text{OUF.id}(5, 6) = \{b\}$
$\text{OUF.id}(2, 3) = \{a, b\}$	



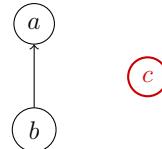
(a) Lorsque la première k -clique $\{1, 2, 3, 4\}$ est traitée par l'algorithme, S est vide, donc un nœud a est créé dans l'OUF, et il est ajouté dans la liste associée à chacune de ses z -cliques. Il en est de même pour la k -clique $\{2, 3, 5, 6\}$, associée au nœud b . Remarquons que la z -clique $\{2, 3\}$ appartient à ces deux k -cliques, donc à ces deux communautés. Elle pointe donc à la fois vers le noeud a et le nœud b .

$\text{OUF.id}(1, 2) = \{a\}$	$\text{OUF.id}(2, 5) = \{a, b\}$
$\text{OUF.id}(1, 3) = \{a\}$	$\text{OUF.id}(2, 6) = \{b\}$
$\text{OUF.id}(1, 4) = \{a\}$	$\text{OUF.id}(3, 5) = \{a, b\}$
$\text{OUF.id}(2, 4) = \{a\}$	$\text{OUF.id}(3, 6) = \{b\}$
$\text{OUF.id}(3, 4) = \{a\}$	$\text{OUF.id}(5, 6) = \{b\}$
$\text{OUF.id}(2, 3) = \{a, b\}$	$\text{OUF.id}(4, 5) = \{a\}$



(b) Lorsque la k -clique $\{2, 3, 4, 5\}$ est traitée, le `Find` de la ligne 6 détecte que toutes les z -cliques de $\{2, 3, 4\}$ sont dans a , et toutes les z -cliques de $\{2, 3, 5\}$ sont dans b . Ainsi, S contient a et b , et ces deux communautés sont fusionnées. Puis, la ligne 12 ajoute a à l'ensemble associé à chacune des z -cliques de $\{2, 3, 4, 5\}$. Il y a alors des redondances $\{a, b\}$ pour les z -cliques $\{2, 3\}$, $\{2, 5\}$ et $\{3, 5\}$, qui seront nettoyées par le prochain `Find` les concernant.

$\text{OUF.id}(1, 2) = \{a\}$	$\text{OUF.id}(2, 5) = \{a, b\}$
$\text{OUF.id}(1, 3) = \{a\}$	$\text{OUF.id}(2, 6) = \{b\}$
$\text{OUF.id}(1, 4) = \{a\}$	$\text{OUF.id}(3, 5) = \{a, b\}$
$\text{OUF.id}(2, 4) = \{a\}$	$\text{OUF.id}(3, 6) = \{b\}$
$\text{OUF.id}(3, 4) = \{a\}$	$\text{OUF.id}(5, 6) = \{b, c\}$
$\text{OUF.id}(2, 3) = \{a, b\}$	$\text{OUF.id}(4, 5) = \{a\}$
$\text{OUF.id}(6, 7) = \{c\}$	$\text{OUF.id}(5, 7) = \{c\}$
$\text{OUF.id}(6, 8) = \{c\}$	$\text{OUF.id}(5, 8) = \{c\}$
$\text{OUF.id}(7, 8) = \{c\}$	



(c) Enfin, le traitement de la dernière k -clique $\{5, 6, 7, 8\}$ crée une nouvelle communauté c , et l'ajoute à chacune de ses z -cliques. On remarque que la z -clique $\{5, 6\}$ appartient aux deux communautés de ce graphe.

FIGURE 3.7 – Exemple du déroulé de l'algorithme 3.2 CPMZ, avec $k = 4$ et $z = 2$, sur le graphe de la figure 3.1. Ce graphe contient quatre 4-cliques : $\{1, 2, 3, 4\}$, $\{2, 3, 4, 5\}$, $\{2, 3, 5, 6\}$ et $\{5, 6, 7, 8\}$, que l'on suppose traitées dans cet ordre dans cet exemple. Les communautés obtenues sont les mêmes avec l'algorithme CPMZ qu'avec l'algorithme CPM : $\{1, 2, 3, 4, 5, 6\}$ et $\{5, 6, 7, 8\}$.

Algorithme 3.2: Algorithme CPMZ.

Entrée: Graphe $G = (V, E)$; $k \in \llbracket 3, +\infty \rrbracket$.
Sortie: OUF représentant les communautés CPMZ de G .

```

1 OUF ← structure d'Overlapping Union-Find vide
2 for each  $k$ -clique  $C_k$  de  $G$  do
3    $S \leftarrow \emptyset$ 
4   for each  $u \in C_k$  do
5      $C_{k-1} \leftarrow C_k \setminus \{u\}$ 
6      $S \leftarrow S \cup \bigcap_{\substack{z\text{-clique } C_z \\ \text{de } C_{k-1}}} \text{OUF.Find}(C_z)$ 
7   if  $S \neq \emptyset$  then
8      $q \leftarrow \text{OUF.Union}(S)$ 
9   else
10     $q \leftarrow \text{OUF.MakeSet}()$ 
11   for each  $z$ -clique  $C_z$  de  $C_k$  do
12      $\text{OUF.Add}(C_z, q)$ 
```

3.7 Analyse

3.7.1 Validité de l'algorithme CPMZ

Le théorème 3.1 suivant montre que les communautés que l'on obtient par l'algorithme CPMZ sont une aggrégation des communautés CPM.

Théorème 3.1 (Validité de l'algorithme CPMZ). *L'algorithme CPMZ renvoie un ensemble de communautés CPMZ. Chaque communauté CPM est incluse dans une et une seule communauté CPMZ.*

Démonstration. Si deux k -cliques C_k^1 et C_k^2 sont adjacentes, cela signifie qu'elles partagent une $(k-1)$ -clique C_{k-1} . Ceci sera correctement détecté par les `Find` de la ligne 6 de l'algorithme 3.2. Pour justifier cela, considérons que C_k^1 est traité avant C_k^2 par l'algorithme. Alors, après l'itération sur C_k^1 dans la boucle principale, en particulier toutes les z -cliques de C_{k-1} appartiennent à un ensemble commun de l'Overlapping Union-Find. La racine de cet ensemble appartient à S pendant l'itération sur C_k^2 , du fait de l'intersection de la ligne 6 au moment de traiter C_{k-1} . Ceci garantit donc l'union des communautés de C_k^2 à celle de C_k^1 , et à la fin de l'algorithme les sommets de ces deux k -cliques sont dans la même communauté.

En d'autres termes, les communautés CPM ne sont jamais divisées par l'algorithme CPMZ et chaque communauté CPM appartient à une seule communauté aggrégée CPMZ (mais une communauté CPMZ peut contenir plus d'une communauté

CPM).

□

3.7.2 Complexité de l'algorithme CPMZ

Pour chaque k -clique C_k , d'après les complexités données dans la section 3.5, l'opération `OUF.Union(S)` de la ligne 8 s'exécute en $\mathcal{O}(|S|)$, le `OUF.MakeSet()` de la ligne 10 en $\mathcal{O}(1)$ et la boucle de la ligne 11 en $\mathcal{O}\left(\binom{k-1}{z}\right)$. Donc, le nombre total d'opérations réalisé par l'algorithme 3.2 est dominé par le nombre d'opérations réalisées au sein des `OUF.Find` de la ligne 6 : la boucle effectue $\binom{k-1}{z}$ appels à `OUF.Find` pour chacune des $(k-1)$ -cliques de C_k , soit $(k-1)\binom{k-1}{z}$ appels à `OUF.Find` pour C_k .

Au total au cours de l'algorithme 3.2, la fonction `OUF.Find` est donc exécutée $n_k \cdot k \cdot \binom{k-1}{z}$ fois, et chacun de ses appels, à la ligne 6, correspondant à la z -clique C_z , s'exécute en $\mathcal{O}(z \cdot |\text{OUF.id}(C_z)|)$. Le facteur z correspond à la recherche de la z -clique dans `OUF.id`. Le facteur $|\text{OUFid}(C_z)|$ correspond au nombre de noeuds de l'`OUF` vers lesquels la z -clique pointe. Il dépend de la z -clique considérée et varie au cours de l'exécution de l'algorithme : il peut soit augmenter lorsque de nouveaux noeuds de l'Overlapping Union-Find sont ajoutés à `OUF.id`(C_z) (à la ligne 12), soit diminuer par les opérations `OUF.Find` (voir figure 3.6 (3.6d)). Ce nombre $|\text{OUF.id}(C_z)|$ est borné par le nombre de k -cliques auxquelles appartient chaque z -clique, qui peut théoriquement être très élevé. Cependant, nous avons calculé en pratique sa moyenne pour chaque `Find` de la ligne 6 sur l'ensemble du déroulé de l'algorithme, et nous montrons dans la section 3.8.2 qu'il ne dépasse jamais 6 dans nos expériences y compris pour les grands graphes, et qu'il est même compris entre 1 et 2 dans la majorité des expériences.

Ainsi, la principale différence dans le temps d'exécution par rapport à l'algorithme CPM exact est le facteur supplémentaire $z \cdot \binom{k-1}{z}$ qui remplace un facteur k de la complexité de CPM. Cela implique que l'algorithme CPMZ a une moins bonne complexité temporelle que l'algorithme CPM, d'au moins ce facteur.

En ce qui concerne l'espace requis, l'algorithme CPMZ doit stocker toutes les z -cliques du graphe qui sont incluses dans une de ses k -cliques. Ce stockage nécessite un espace en $\mathcal{O}(z \cdot n_z)$ (où n_z est le nombre de z -cliques du graphe), bien plus faible que pour le stockage des $(k-1)$ -cliques, en $\mathcal{O}((k-1) \cdot n_{k-1})$. De plus, il faut tenir compte du nombre maximal de noeuds de l'`OUF` attribué à chaque z -clique au cours de l'algorithme. En théorie, ce nombre augmente la complexité en mémoire. Cependant, nous montrons dans la Section 3.8 qu'en pratique, les besoins en mémoire de l'algorithme CPMZ sont bien inférieurs à ceux de l'algorithme CPM.

Enfin, il convient de noter que l'algorithme CPMZ nécessite un post-traitement pour obtenir l'ensemble des sommets de chaque communauté. Ce post-traitement itère sur toutes les z -cliques, pour ajouter ses sommets dans ses communautés enregistrées par l'`OUF`. Néanmoins, dans le cas qui nous intéresse (les graphes massifs du

monde réel), il y a beaucoup moins de z -cliques que de k -cliques. Ce post-traitement est donc beaucoup moins coûteux que le reste de l'algorithme.

3.8 Évaluation expérimentale

Nous avons réalisé une étude expérimentale de nos implémentations de CPM et de CPMZ, afin d'en évaluer les performances en termes de temps de calcul et d'utilisation mémoire. Nous commençons par comparer notre implémentation de CPM à celles qui existent dans l'état de l'art, puis nous comparons cette implémentation avec celle de CPMZ pour montrer les avantages que l'on peut tirer de l'algorithme CPMZ. Enfin, nous montrons que les communautés CPMZ sont proches des communautés CPM.

Matériel. Nous avons réalisé nos expériences sur une machine Linux DELL PowerEdge R440, équipée de 2 processeurs Intel Xeon Silver 4216 à 32 cœurs chacun, et de 390 Go de RAM.

Jeux de données. Nous réalisons nos expériences sur plusieurs graphes massifs issus du monde réel, que nous avons obtenus à partir de [LK14]. Leurs caractéristiques sont présentées dans le tableau 3.1, où ils sont triés par dégénérescence croissante. La dégénérescence traduit le caractère dense du graphe, qui a un impact sur le nombre de k -cliques. Plus elle est élevée et plus il est difficile d'obtenir les communautés CPM dans ces jeux de données.

Graphe	n	m	c	$k_{min} - k_{max}$	$n_{k_{min}}$	$n_{k_{max}}$
<i>Soc-Pokec</i>	1 632 803	22 031 964	47	3 - 15	32 557 458	353 958 854
<i>Loc-Gowalla</i>	196 591	950 327	51	3 - 15	2 273 138	201 454 150
<i>YouTube</i>	1 134 890	2 987 624	51	3 - 15	3 056 386	1 068
<i>Zhishi-Baidu</i>	2 140 198	17 014 946	78	3 - 15	25 207 196	1 080 702 188
<i>AS-Skitter</i>	1 696 415	11 095 298	111	3 - 6	28 769 868	9 759 000 981
<i>DBLP</i>	425 957	1 049 866	113	3 - 7	2 224 385	60 913 718 813
<i>WikiTalk</i>	2 394 385	4 659 565	131	3 - 7	9 203 519	5 490 986 046
<i>Orkut</i>	3 072 627	117 185 083	253	3 - 5	627 584 181	15 766 607 860
<i>Friendster</i>	124 836 180	1 806 067 135	304	3 - 4	4 173 724 124	8 963 503 236
<i>LiveJournal</i>	4 036 538	34 681 189	360	3 - 4	177 820 130	5 216 918 441

TABLEAU 3.1 – Notre ensemble de graphes issus du monde réel, classés par dégénérescence c , où n est le nombre de sommets et m le nombre d'arêtes. k_{min} et k_{max} représentent les valeurs de k minimale et maximale sur lesquelles nous avons pu exécuter notre implémentation CPM. n_k est le nombre de k -cliques du graphe. Ces graphes peuvent être directement téléchargés depuis la base de données SNAP [LK14]

Implémentation. Pour l’implémentation de CPM, nous substituons l’un des meilleurs algorithmes d’énumération des k -cliques [DBS18] à celui utilisé dans l’implémentation de Kumpula *et al.* [Kum+08]. Nos implémentations de CPM et CPMZ ont été faites en langage C. Elles sont disponibles en libre accès³.

Pour nous comparer à l’état de l’art, nous avons utilisé les implémentations publiques des algorithmes [GLM13; Kum+08; Pal+05; RMH12]. Nous rappelons que l’état de l’art travaille à partir de l’énumération des cliques maximales, et non pas des k -cliques, à l’exception de Kumpula *et al.* [Kum+08] dont nous améliorons l’implémentation.

Protocole expérimental. Pour chaque graphe et chaque algorithme, nous avons exécuté les algorithmes de CPM pour toutes les valeurs de k allant de 3 à 15. L’exécution a été stoppée lorsque le temps de calcul a dépassé 72 heures ou que l’utilisation de la RAM a dépassé 390 Go.

Nous avons exécuté l’algorithme CPMZ pour $z = 2$ et $z = 3$. Il a pu être calculé dans tous les cas pour lesquels CPM fonctionne, sauf pour $z = 3$ dans les graphes *Zhishi-Baidu* avec $k = 15$ et *DBLP* avec $k = 7$.

Il est par ailleurs intéressant de constater que nous parvenons à obtenir des résultats avec l’implémentation de CPMZ dans les cas où le calcul n’a pas pu être effectué par l’algorithme CPM (voir la figure 3.9).

3.8.1 Comparaison de notre implémentation CPM à l’état de l’art

Dans cette section, nous comparons notre implémentation de CPM à celles qui existent dans l’état de l’art.

L’algorithme proposé par Palla *et al.* dans l’article original introduisant CPM [Pal+05] est quadratique en nombre de cliques maximales. Étant donné que nous nous intéressons à des graphes comportant au moins plusieurs millions de cliques, l’algorithme ne peut pas les traiter en des temps raisonnables. Nous ne réalisons donc pas d’expériences avec cet algorithme. Par ailleurs, nos tests ont montré que l’algorithme de Reid *et al.* [RMH12] a de meilleures performances que celui de Gregori *et al.* [GLM13] (version séquentielle) dans tous les cas ; nous ne présentons donc pas les résultats obtenus avec la version de Gregori *et al.* Nous comparons donc ici notre implémentation avec celles de Reid *et al.* [RMH12] et de Kumpula *et al.* [Kum+08].

La figure 3.8 compare le temps (3.8 (a)) et la mémoire (3.8 (b)) nécessaires au calcul des communautés CPM par notre implémentation de CPM et ces algorithmes

3. <https://gitlab.lip6.fr/baudin/cpm-cpmz>

de l'état de l'art. Pour chaque algorithme de l'état de l'art, nous traçons son temps d'exécution (*resp.* l'utilisation de la mémoire) divisé par le temps d'exécution (*resp.* l'utilisation de la mémoire) de notre algorithme CPM. Une valeur au-dessus de $y = 1$ signifie que le temps d'exécution est plus lent que notre implémentation, et en dessous qu'il est plus rapide.

Nous remarquons tout d'abord que l'implémentation de Kumpula *et al.* est systématiquement moins efficace que la nôtre, comme attendu, et que dans la plupart des cas celle de Reid *et al* l'est aussi.

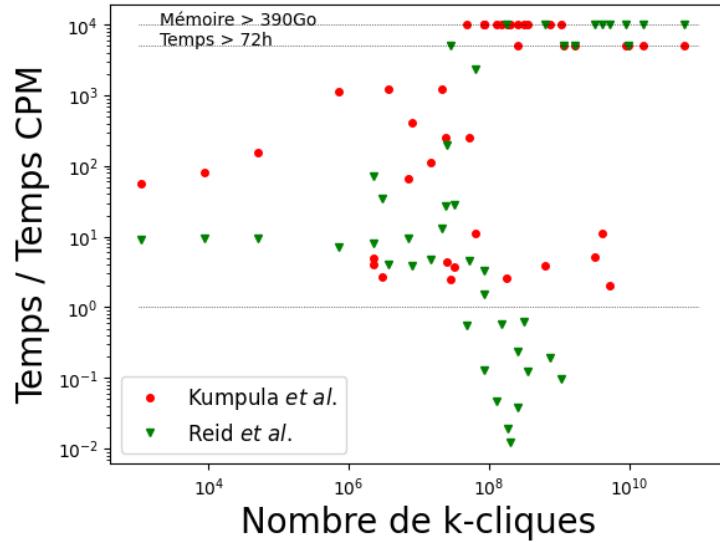
Néanmoins, l'algorithme de Reid *et al.* est plus efficace que le nôtre dans certaines configurations : lorsque k devient trop élevé, sur nos petits jeux de données, les points verts qui sont directement sous la ligne $y = 1$ correspondent en effet aux graphes *Soc-Pokec*, *Loc-Gowalla*, *YouTube* et *Zhishi-Baidu*, pour les plus grandes valeurs de k . En outre, le graphe *DBLP*, qui est un graphe de co-auteurs scientifiques, a ses sommets organisés en grandes cliques du fait de la nature de ce graphe, ce qui rend l'énumération des cliques maximales beaucoup plus efficace que celle des k -cliques : l'algorithme de Reid *et al.* parvient à calculer les communautés en 10 secondes alors qu'il faut plusieurs heures pour traiter le grand nombre de k -cliques avec notre algorithme CPM.

Cependant, cet algorithme ne permet pas de traiter les plus grands graphes de nos jeux de données. Le graphe intermédiaire *AS-Skitter* contient trop de cliques et l'algorithme ne fournit pas de résultat en moins de 72 heures. Pour les graphes plus denses (*WikiTalk*, *Orkut*, *Friendster*, *LiveJournal*), il y a trop de cliques maximales pour qu'elles puissent être stockées en mémoire vive et l'algorithme ne peut pas fonctionner avec moins de 390 Go de RAM, alors que le nôtre est capable de calculer le résultat.

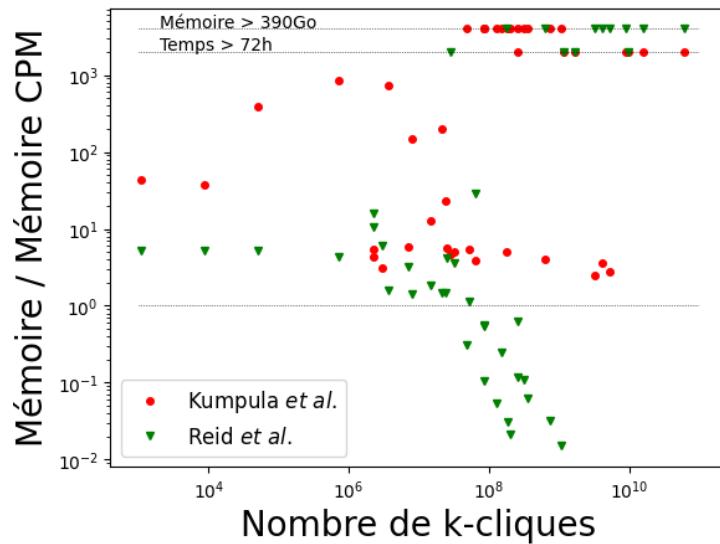
Concernant la consommation mémoire, on observe globalement les mêmes rapports que pour les temps de calcul. On en déduit notamment que les cas où Reid *et al.* mettent moins de temps à calculer les communautés proviennent des graphes pour lesquels il y a beaucoup moins de cliques maximales de tailles supérieures à k que de k -cliques.

Enfin, nous constatons que notre implémentation de CPM est capable d'obtenir un ensemble de communautés dans des cas où aucune autre ne peut en fournir. Ces cas sont représentés par les points sur les deux lignes horizontales en haut de la figure de la figure 3.8 (a) (et 3.8 (b)). Chaque point sur ces lignes correspond à un graphe et une valeur de k pour lesquels le calcul n'a pas terminé dans les limites de temps et de mémoire que nous avons fixé.

Nous sommes capables de traiter des graphes contenant jusqu'à 10^{11} k -cliques, là où Reid *et al.* atteignent 10^9 du fait de la difficulté d'énumérer les cliques maximales quand le graphe est trop massif, et Kumpula *et al.* atteignent 10^{10} . Nous gagnons



(a) Temps d'exécution de notre implémentation de CPM divisé par le temps d'exécution des deux algorithmes de l'état de l'art auxquels nous nous comparons.



(b) Consommation mémoire de notre implémentation de CPM divisée par la consommation mémoire des deux algorithmes de l'état de l'art auxquels nous nous comparons.

FIGURE 3.8 – Comparaison entre notre implémentation CPM et celles de l'état de l'art. Chaque point correspond à une exécution sur un graphe de notre ensemble de jeux de données et un k donné. Nous affichons les résultats en fonction de n_k , où n_k est le nombre de k -cliques du graphe d'entrée. Le temps maximal d'exécution est limité à 72h et la mémoire maximale à 390 Go. Un marqueur placé sur la ligne correspondante indique donc un calcul qui ne s'est pas terminé, en raison d'une limite de temps ou de mémoire. La ligne $y = 1$ est tracée pour montrer les cas moins efficaces que notre implémentation (au-dessus), et ceux qui sont plus efficaces (en-dessous).

donc un ordre de grandeur avec notre implémentation de CPM, grâce à l'énumération efficace des k -cliques que nous exploitons. Cela confirme l'intérêt de travailler avec les k -cliques, même si nous y perdons en temps de calcul et en mémoire consommée, dans certains cas où les cliques maximales sont plus avantageuses.

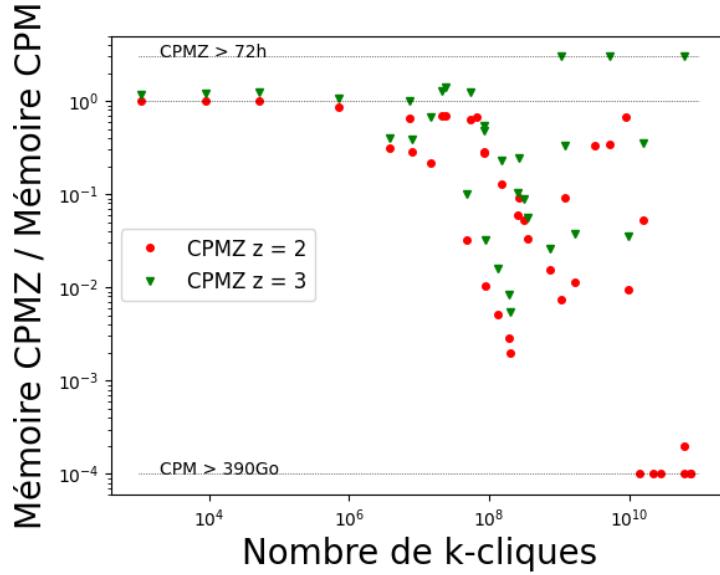
3.8.2 Étude du gain en mémoire et du coût en temps de l'algorithme CPMZ

L'algorithme CPMZ est un compromis entre mémoire et temps : son temps d'exécution plus élevé que celui de CPM, mais il nécessite moins de mémoire. La figure 3.9 présente ces deux aspects sur nos jeux de données à partir de nos implementations de CPM et de CPMZ.

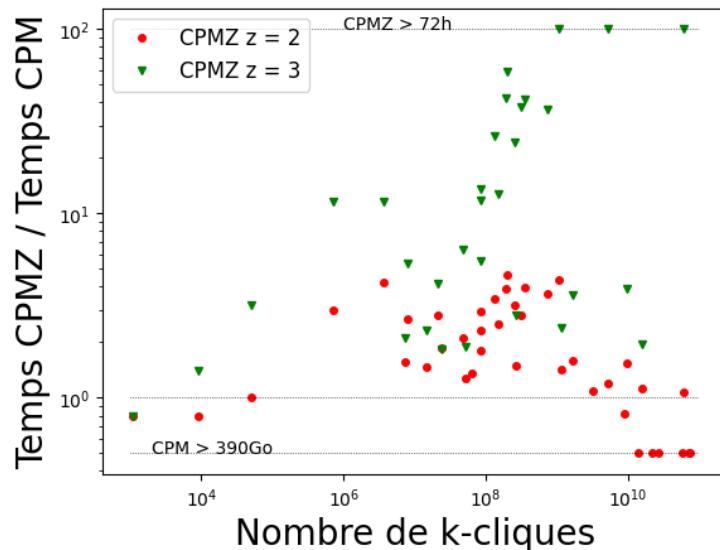
La figure 3.9 (a) compare la mémoire utilisée par nos algorithmes CPM, et CPMZ avec $z = 2$ et $z = 3$. Nous présentons la mémoire utilisée par CPMZ divisée par la mémoire utilisée par CPM, en fonction du nombre de k -cliques du graphe pour la valeur de k considérée. Comme pour la figure 3.8, nous représentons les cas où CPMZ dépasse la limite de temps sur une ligne horizontale en haut des figures. De plus, les cas où nous obtenons des résultats avec CPMZ et non avec CPM sont représentés sur une ligne horizontale en bas des figures. Notez que pour certains petits graphes, le nombre de $(k - 1)$ -cliques qui sont stockées par CPM est inférieur au nombre de z -cliques. Pour ces graphes, CPMZ nécessite plus de mémoire que CPM. Dans la plupart des cas cependant, et notamment pour les graphes les plus massifs, nous observons un gain de mémoire considérable, et dans certains cas, il est même possible d'obtenir un ensemble de communautés là où notre algorithme CPM n'en fournit pas (et l'état de l'art non plus). Ces cas sont représentés par les points en bas à droite de la figure, sur la ligne qui indique que le calcul de CPM ne s'est pas terminé à cause d'un trop gros besoin en mémoire. Ils correspondent aux graphes *AS-Skitter* avec $k = 7$, *WikiTalk* avec $k = 8, 9$, *Orkut* avec $k = 6$ et *Friendster* avec $k = 5, 6$.

En ce qui concerne la durée d'exécution de l'algorithme CPMZ, la figure 3.9 (b) la compare à celle de notre implémentation CPM. On constate que dans quasiment tous les cas, le temps de calcul de CPM est plus rapide que celui de CPMZ, allant jusqu'à un facteur de presque 100. Dans certains rares cas, CPMZ est plus légèrement plus rapide que CPM : c'est le cas sur les graphes avec très peu de k -cliques, pour des raisons d'implémentation, et pour les graphes avec beaucoup de k -cliques, pour lesquels le post-traitement de CPMZ sur toutes les $(k - 1)$ -cliques peut être beaucoup plus coûteux que celui de CPMZ sur les z -cliques, ce qui peut se ressentir sur les temps de calcul.

Comme indiqué dans la section 3.7, deux facteurs font que CPMZ est plus lent que CPM. Le premier est le fait qu'une z -clique peut pointer vers plusieurs noeuds de l'Overlapping Union-Find. Pour comprendre l'importance de ce facteur, nous effectuons des expériences pour évaluer le temps associé au parcours des noeuds as-



(a) Consommation mémoire de l'algorithme CPMZ, pour $z = 2$ et $z = 3$, divisés par la consommation mémoire de notre implémentation CPM, en fonction du nombre de k -cliques du graphe n_k .



(b) Temps d'exécutions de l'algorithme CPMZ, pour $z = 2$ et $z = 3$, divisé par le temps d'exécution de notre implémentation CPM, en fonction du nombre de k -cliques du graphe n_k .

FIGURE 3.9 – Comparaison entre le temps et la consommation de mémoire de l'algorithme CPMZ et de notre algorithme CPM. Le temps maximal d'exécution est limité à 72h, et la mémoire maximale n'est pas limitée, car elle est toujours inférieure à celle requise par CPM. Un marqueur placé sur la ligne horizontale en haut de la figure indique donc un calcul qui ne s'est pas terminé, en raison de la limite de temps. Un marqueur placé sur la ligne horizontale en bas de la figure indique un calcul qui se termine pour l'algorithme CPMZ, mais pas pour CPM. La ligne $y = 1$ est tracée pour montrer les cas de CPM qui sont moins efficaces que CPMZ (au-dessus), et ceux qui sont plus efficaces (en-dessous).

sociés aux z -cliques dans l’Overlapping Union-Find, au sein de l’algorithme. Pour ce faire, dans l’algorithme 3.2, nous calculons la somme de la taille de chaque ensemble $\text{OUF.id}(C_z)$ sur lesquels un **Find** est réalisé à la ligne 6, qui est la partie de l’algorithme qui concentre le plus d’opérations. Nous divisons ensuite cette somme par le nombre de fois où cette opération **Find** est effectuée, soit $n_{k-1} \cdot k \cdot \binom{k-1}{z}$. Nous observons alors que ce facteur reste faible : pour tous les graphes et toutes les valeurs de k et z , il est compris entre 1 et 2, à l’exception des petits graphes de *YouTube* avec une valeur autour de 4 pour $z = 2$ et $k \in \llbracket 10, 15 \rrbracket$, $z = 3$ et $k \in \llbracket 11, 15 \rrbracket$, et de *Zhishi-Baidu* avec une valeur autour de 5 pour $z = 2$ et $k = 5, 6, 7$. Ainsi, ce facteur est faible en pratique et ne joue donc pas un rôle important dans le ralentissement du temps d’exécution.

L’autre facteur qui cause un ralentissement du temps de calcul est le facteur supplémentaire $\binom{k-1}{z}$ dans la complexité, induit par le fait que nous traitons pour chaque k -clique, toutes les z -cliques incluses dans ses $(k - 1)$ -clique. Ce facteur est élevé et implique que le temps de calcul limite le calcul de CPMZ, même si la mémoire n’est plus limitante.

3.8.3 Similarités entre communautés CPM et CPMZ

Nous rappelons que les communautés obtenues avec l’algorithme CPMZ correspondent aux communautés CPM dont certaines sont agglomérées entre elles.

Pour mesurer la précision de l’algorithme CPMZ, nous comparons les ensembles de communautés CPMZ aux ensembles de communautés CPM associés. Pour ce faire, nous utilisons une implémentation d’une mesure d’information mutuelle normalisée (NMI), adaptée aux ensembles de communautés qui se chevauchent. Cet outil est fourni par McAid *et al.* [MGH13]. Il permet de mesurer le degré de similitude entre deux ensembles de communautés qui se chevauchent.

Nous avons effectué les comparaisons de similarité entre les communautés CPM et CPMZ sur tous les graphes de nos jeux de données, avec toutes les valeurs de k pour lesquelles nous pouvons calculer les communautés avec l’algorithme CPM (voir le tableau 3.1). Nous observons ce qui suit :

- pour CPMZ avec $z = 2$, la similarité moyenne est de 98.6%, la médiane est de 99.4% et toutes les valeurs sont supérieures à 93.8% ;
- pour CPMZ avec $z = 3$, la similarité moyenne est de 99.95%, la médiane est de 100% et toutes les valeurs sont supérieures à 99.5%.

Cela confirme que les fusions incorrectes entre communautés réalisées par l’algorithme CPMZ par les cliques parasites ont peu d’influence sur le résultat final : la structure des communautés est à peine affectée par l’algorithme CPMZ, les communautés CPMZ sont très proches des communautés CPM.

3.9 Conclusion et discussion

Dans ce chapitre, nous avons abordé le problème de la détection de communautés qui se recouvrent, dans les graphes, par le biais de la méthode de percolation de cliques (CPM). Nos contributions sont doubles. Nous avons d'abord proposé une amélioration de l'implémentation de l'algorithme CPM exact en tirant parti d'une méthode efficace d'énumération de k -cliques. Nous avons ensuite proposé un algorithme heuristique appelé CPMZ qui fournit des communautés agglomérées, c'est-à-dire des communautés qui sont des sur-ensembles des communautés de CPM. Cet algorithme utilise beaucoup moins de mémoire que l'algorithme exact, au prix d'un temps d'exécution plus élevé.

Grâce à des expériences sur un ensemble de graphes provenant de contextes différents, nous montrons que :

- notre implémentation CPM surpassé les algorithmes de l'état de l'art dans de nombreux cas, et nous sommes en mesure de calculer les communautés CPM dans des cas où cela n'était pas possible auparavant. Cependant, elle n'est pas meilleure dans tous les cas, notamment comme on l'a vu pour le graphe *DBLP* pour lequel l'énumération des cliques maximales est plus efficace que celle des k -cliques. Cela soulève la question intéressante de savoir s'il est possible de prédire quelle méthode d'énumération serait la plus efficace sur un graphe, en étudiant au préalable sa structure ;
- notre algorithme approché CPMZ utilise beaucoup moins de mémoire que l'algorithme exact. Même si son temps d'exécution est plus élevé, cela nous permet d'obtenir des communautés agglomérées dans des cas où aucun autre algorithme ne peut fournir de résultat ;
- enfin les résultats fournis par l'algorithme CPMZ sont très proches de ceux de CPM, comme l'atteste la mesure NMI adaptée aux communautés qui se chevauchent [MGH13].

Plusieurs perspectives intéressantes se présentent dans le cadre de ce travail. Il faut noter que dans l'algorithme CPMZ, l'ordre dans lequel les k -cliques sont traitées joue un rôle important dans la fusion de communautés due à des cliques parasites, que nous ne comprenons pas encore tout à fait. Nos expériences montrent qu'en pratique, seules quelques mauvaises fusions de communautés CPM se produisent, ce qui aboutit à un résultat quasi exact. Cela soulève de nombreuses questions intéressantes sur la caractérisation de ces cliques parasites : combien y en a-t-il dans un graphe typique issu du monde réel ? Nous pensons qu'il est possible de construire des exemples dans lesquels aucun ordre de traitement des k -cliques ne conduira à un ensemble de communautés CPM exacte avec l'algorithme CPMZ. Cependant, dans de nombreux cas incluant des graphes issus du monde réel, il est possible qu'un certain ordre de traitement des k -cliques produise des résultats de meilleure qualité que

d'autres ordres. Cela soulève la question de savoir comment concevoir un tel ordre. Une autre possibilité intéressante serait d'exécuter l'algorithme CPMZ avec deux ou plusieurs ordres différents de k -cliques pour raffiner leurs résultats : étant donné que chaque communauté CPMZ est une union de communautés CPM exactes, il serait possible de comparer les communautés CPMZ des deux sorties pour sélectionner une partie des communautés de chaque résultat, qui formerait un ensemble plus proche du résultat exact.

L'algorithme CPM construit les communautés en y ajoutant les k -cliques une par une, à la volée. Cette approche laisse penser qu'il peut être étendu dans les cas de données temporelles, où les k -cliques temporelles arrivent dans le temps à la volée. Cette extension a en réalité été proposée deux ans après la mise en place de la définition de CPM [PBV07]. Nous verrons dans le chapitre 6 un algorithme pour énumérer les communautés CPM dans les flots de liens, qui, comme ici, tire partie d'un nouvel algorithme efficace pour énumérer les k -cliques, ainsi que d'une structure d'Union-Find adaptée à des ensembles de sommets temporels.

Enfin, comme on l'a vu, une partie des algorithmes CPM se base sur l'énumération des cliques maximales du graphe, ce qui fournit certains avantages, même si on a noté le bénéfice de travailler avec l'énumération des k -cliques. Alors, de la même manière que ce que nous avons fait ici, travailler à améliorer l'énumération des cliques maximales pourrait aider à étendre la méthode CPM à des cas où l'énumération des k -cliques serait bloquante, notamment pour faire augmenter la valeur de k pour le calcul des communautés dans les graphes massifs. Dans cette thèse, nous travaillons en particulier sur les problèmes d'énumération des bicliques maximales dans les graphes bipartis (chapitre 4), et des cliques maximales dans les flots de liens (chapitre 5).

Énumération des bicliques maximales dans les grands graphes bipartis peu denses

Résumé

Les graphes bipartis sont très utilisés pour modéliser les réseaux du monde réel car ils permettent de représenter des interactions entre des sommets de deux types différents. Les bicliques sont des ensembles de sommets dont ceux du premier type sont connectés à tous ceux du deuxième type dans cet ensemble. Dans ce chapitre, nous proposons un nouvel algorithme d'énumération des bicliques maximales. Cet algorithme est plus simple que l'algorithme de référence de l'état de l'art, et il permet de réaliser l'énumération sur des graphes massifs impossibles à traiter avec les implémentations existantes. Nous réalisons des expériences pour valider l'efficacité de cet algorithme, et montrons que l'ordre des sommets ainsi que le choix de l'un des deux types de sommets sur lequel lancer l'énumération ont un impact sur le temps de calcul. En particulier, nous constatons que le comportement de l'algorithme n'est pas le même sur les graphes les plus massifs et sur les petits graphes, ce qui est intéressant dans l'étude des données issues du monde réel.

4.1 Introduction

Les graphes bipartis sont largement utilisés pour représenter les réseaux du monde réel [GL06]. Ils peuvent modéliser de nombreux systèmes dans lesquels les entités qui interagissent sont de deux types différents, tels que les plateformes en ligne où des utilisateurs qui sélectionnent du contenu (regarder des vidéos, cliquer sur des liens, acheter un produit, *etc.*), ou des individus qui collaborent à des projets, ou encore des personnes qui participent à des événements. Ils peuvent également représenter des données d'interaction dans d'autres domaines, comme biologiques avec les gènes qui codent des protéines et les protéines qui inhibent ou stimulent l'activité du génome.

Comme pour les graphes monopartis (non bipartis), l'identification de sous-graphes denses dans ces réseaux a suscité beaucoup d'intérêt. Elle révèle par exemple des utilisateurs ayant des intérêts communs ou des protéines liées à la même région de la molécule d'ADN. De manière similaire aux graphes monopartis, une biclique est un sous-graphe de densité maximale dans les réseaux bipartis : c'est un sous-graphe dans lequel toutes les entités du premier type sont connectées à toutes les entités du deuxième type.

La détection des bicliques, en particulier dans les grands graphes, a fait l'objet de nombreux travaux. En particulier, récemment, Chen *et al.* ont introduit un algorithme pour énumérer les bicliques maximales, beaucoup plus efficace que les autres algorithmes existants, en adaptant l'algorithme de Bron-Kerbosch [BK73].

Ce chapitre développe un nouvel algorithme pour résoudre ce problème, plus simple et plus efficace que l'état de l'art. Il est organisé de la manière suivante :

- Dans la section 4.2, nous présentons l'état de l'art de l'énumération des cliques maximales dans les graphes (monopartis). Nous commençons par présenter en détail les algorithmes de référence pour traiter des jeux de données massifs issus du monde réel : l'algorithme de Bron-Kerbosch [BK73] et son amélioration par Eppstein *et al.* [ELS10] Ces algorithmes sont les points de départ des travaux de ce chapitre 4 (graphes bipartis) et du chapitre 5 (flots de liens). Après avoir présenté ces algorithmes, nous présentons le reste de l'état de l'art de l'énumération de cliques maximales dans les graphes.
- Dans la section 4.3 nous présentons l'état de l'art de l'énumération des bicliques maximales dans les graphes bipartis, et notamment les travaux de Chen *et al.* qui surpassent de loin les travaux précédents.
- Dans la section 4.4, nous introduisons un nouvel algorithme pour énumérer les bicliques maximales, plus simple, et nous faisons une analyse de sa complexité.
- La section 4.5 présente les expériences que nous avons réalisées pour valider notre algorithme sur des jeux de données massifs issus du monde réel. Pour cela, nous en avons fait une implémentation en C++, disponible en ligne¹. Nous montrons expérimentalement que notre algorithme est beaucoup plus efficace sur les jeux de données massifs, et qu'il permet de faire l'énumération là où l'état de l'art ne passe pas à l'échelle. Puis, nous discutons de l'impact du choix du côté du graphe biparti sur lequel réaliser l'énumération, et de l'ordre des sommets sur le temps de calcul, puis nous montrons que notre algorithme est plus consommateur en mémoire que l'état de l'art, mais que ça n'est pas le facteur limitant de l'énumération.
- Enfin, la section 4.6 conclut le chapitre et en présente les perspectives.

1. <https://gitlab.lip6.fr/baudin/bbk>

4.2 Énumération des cliques maximales dans les graphes

Nous rappelons que dans un graphe, une clique est un ensemble de sommets tous connectés entre eux, et qu'elle est maximale si elle n'est incluse dans aucune autre clique. La figure 4.1 donne un exemple d'un graphe, dans lequel les cliques maximales sont $\{1, 2, 3\}$, $\{2, 3, 4, 5\}$, $\{4, 5, 7\}$ et $\{4, 6, 7\}$.

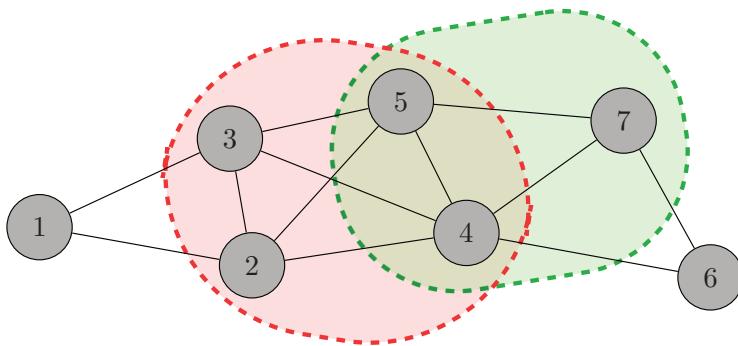


FIGURE 4.1 – Exemple d'un graphe où les deux cliques maximales qui contiennent le sommet 5 sont entourées en couleur : $\{2, 3, 4, 5\}$ et $\{4, 5, 7\}$.

Comme nous l'avons vu, il est important de pouvoir extraire des sous-graphes très denses, qui peuvent être une clé pour la compréhension de leur structure. Or, ce problème est connu pour être NP-difficile, ce qui pose un défi pour sa mise en œuvre en pratique [LLR80]. Des algorithmes d'énumération efficaces ont été conçus pour traiter des graphes massifs représentant des réseaux d'interactions issues du monde réel. En particulier, l'algorithme de Bron-Kerbosch [BK73] est utilisé sur des instances larges et peu denses, notamment grâce à l'utilisation d'un pivot, qui permet de réduire considérablement l'espace de recherche [TTT06]. Eppstein *et al.* [ELS10] adaptent cet algorithme en utilisant une méthode d'ordonnancement des nœuds adéquate et une implémentation efficace qui permet de passer à l'échelle de grands graphes peu denses issus du monde réel.

Dans ce qui suit, nous présentons l'algorithme de Bron-Kerbosch avec son pivot [BK73], puis nous présentons l'algorithme d'Eppstein *et al.* [ELS10] qui permet de tirer profit de l'ordonnancement des sommets du graphe, ainsi que leurs complexités associées. Ce sont les travaux de référence pour l'énumération des cliques maximales, et nous nous en servirons comme base pour ce chapitre et le suivant. Ensuite, nous présentons le contexte actuel de la recherche dans ce domaine, avec les autres travaux récents sur l'énumération des cliques maximales.

4.2.1 Algorithmes de Bron-Kerbosch et Eppstein *et al.*

L'algorithme de référence pour énumérer les cliques maximales dans les graphes du monde réel est celui publié par Eppstein *et al.* [ELS10]. Il s'agit d'une adaptation du célèbre algorithme de Bron-Kerbosch [BK73]. En utilisant cet algorithme, Eppstein *et al.* ont conçu une méthode permettant de tirer parti de l'ordre des sommets dans V , pour améliorer la complexité de l'énumération de Bron-Kerbosch.

Tout d'abord, nous présentons l'algorithme 4.1, conçu par Bron et Kerbosch [BK73]. Il s'agit d'un algorithme récursif, qui construit les cliques maximales en ajoutant leurs sommets un par un. Un appel à $\text{BK}(R, P, X)$ prend pour argument une clique en construction R et l'ensemble des sommets qui peuvent faire grandir la clique R , qui sont répartis entre P et X . En d'autres termes, $P \cup X$ contient tous les voisins communs des sommets de R , *i.e.* $P \cup X = \bigcap_{u \in R} N(u)$ (voisinage de la clique R).

Ainsi, une clique est maximale lorsque $P \cup X = \emptyset$. De plus, la partition entre P et X est telle que P correspond à l'ensemble des sommets candidats **à utiliser** pour agrandir la clique R , et X est l'ensemble des sommets qui peuvent agrandir la clique R , mais qui ont déjà été traités, et qui **ne doivent donc pas être utilisés**, pour ne pas renvoyer de clique déjà énumérée. Ainsi, un appel à $\text{BK}(R, P, X)$ énumère l'ensemble des cliques maximales de G contenant R , certains sommets de P et aucun sommet de X . Alors, l'appel $\text{BK}(\emptyset, V, \emptyset)$ énumère l'ensemble des cliques maximales du graphe (ligne 8). Notez que si $P = \emptyset$, mais $X \neq \emptyset$, alors R n'est pas maximale et la recherche récursive s'arrête. Dans ce cas, l'algorithme a parcouru une branche de l'arbre d'appel récursif qui ne mène pas à une clique maximale. Ces branches inutiles diminuent l'efficacité du calcul, ce qui conduit à l'introduction d'un pivot, qui cherche à réduire leur nombre.

Algorithme 4.1: Algorithme de Bron-Kerbosch [BK73].

Entrée: Graphe $G = (V, E)$.

Sortie: Ensemble des cliques maximales de G .

1 Fonction $\text{BK}(R, P, X)$:

```

2 if  $P \cup X = \emptyset$  then
3   output  $R$  clique maximale
4 for  $u \in P$  do
5   BK( $R \cup \{u\}$ ,  $P \cap N(u)$ ,  $X \cap N(u)$ )
6    $P \leftarrow P \setminus \{u\}$ 
7    $X \leftarrow X \cup \{u\}$ 

```

8 BK(\emptyset, V, \emptyset) // Appel initial

L'algorithme 4.2, également proposé par Bron et Kerbosch, utilise un pivot. Ce pivot supprime une partie des branches de l'arbre des appels récursifs qui ne

conduisent à aucune clique maximale. Le principe est le suivant : dans un appel récursif de BK donné avec les ensembles R , P et X , si p est un sommet qui peut faire croître la clique R en une clique $R \cup \{p\}$ (*i.e.* $p \in P \cup X$), alors toute clique contenant R contient soit p , soit un sommet qui ne peut pas être dans une clique avec p , c'est-à-dire qu'il n'est pas voisin de p . On peut donc réduire le choix des sommets à utiliser pour faire grandir R soit à p , soit à un sommet qui n'est pas voisin de p . C'est ce qui est mis en œuvre par les lignes 4 et 5 de l'algorithme 4.2. L'enjeu est alors de sélectionner le pivot $p \in P \cup X$ qui est le plus efficace en termes de gain dans l'espace de recherche. Tomita *et al.* [TTT06] ont montré que la méthode qui fonctionne le mieux consiste à choisir le sommet p qui maximise le nombre d'appels récursifs éliminés, c'est-à-dire qui maximise $|P \cap N(p)|$. Autrement dit, on choisit $p = \underset{u \in P \cup X}{\operatorname{argmax}} (|P \cap N(u)|)$. Tomita *et al.* ont montré que cet algorithme s'exécute en $\mathcal{O}(3^{n/3})$, où n est le nombre de sommets du graphe.

Algorithme 4.2: Algorithme de Bron-Kerbosch avec pivot [BK73].

Entrée: Graphe $G = (V, E)$.

Sortie: Ensemble des cliques maximales de G .

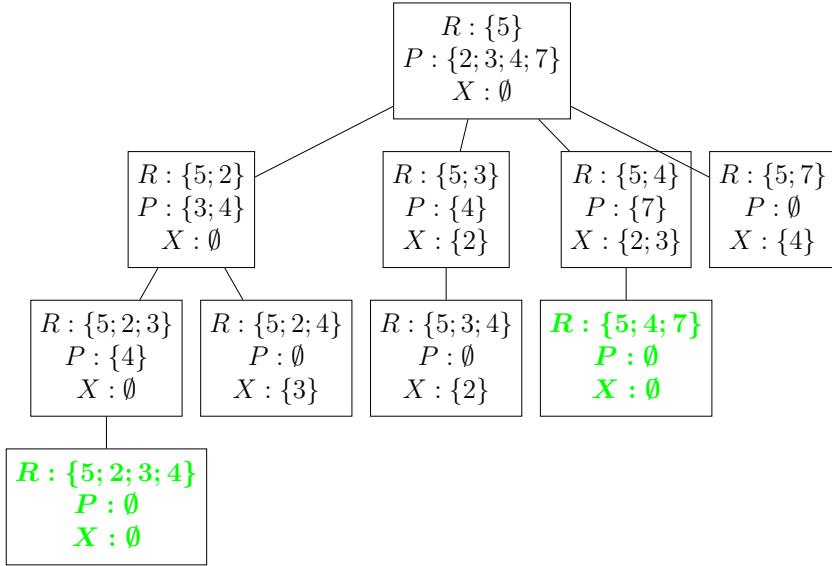
```

1 Fonction BKPivot( $R, P, X$ ):
2   if  $P \cup X = \emptyset$  then
3     output  $R$  clique maximale
4   p  $\leftarrow$  choisir un pivot dans  $P \cup X$ 
5   for  $u \in P \setminus N(p)$  do
6     BKPivot( $R \cup \{u\}, P \cap N(u), X \cap N(u)$ )
7      $P \leftarrow P \setminus \{u\}$ 
8      $X \leftarrow X \cup \{u\}$ 
9 BKPivot( $\emptyset, V, \emptyset$ )                                // Appel initial

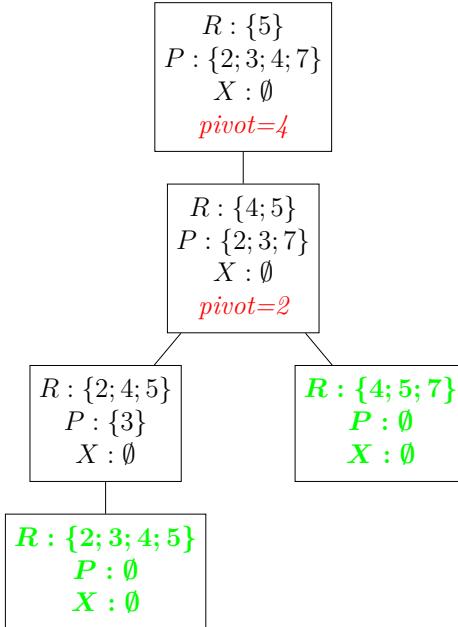
```

La figure 4.2 donne un exemple de l'exécution de l'algorithme de Bron-Kerbosch avec et sans pivot, sur le sommet 5 du graphe de la figure 4.1. Grâce au pivot, le nombre de noeuds de l'arbre des appels est divisé par deux.

L'algorithme 4.3, introduit par Eppstein *et al.* [ELS10], est la version de référence pour l'énumération des cliques maximales d'un graphe. Il traite les sommets un par un, **dans un ordre bien choisi**. Pour chaque sommet v_i , il construit l'ensemble P des voisins de v_i qui n'ont pas encore été traités (ligne 2), et l'ensemble X des voisins de v_i qui ont déjà été traités (ligne 3). Ensuite, à la ligne 4, il appelle la fonction BKPivot, décrite dans l'algorithme 4.2, qui énumère l'ensemble des cliques maximales qui contiennent v_i , des sommets de P , et aucun sommet de X . Ainsi, pour chaque sommet, cet algorithme énumère toutes les cliques maximales qui contiennent ce sommet et aucun des sommets déjà traités. Chaque clique maximale est donc énumérée exactement une fois.



(a) Arbre des appels pour l'algorithme de Bron-Kerbosch *sans pivot* exécuté sur son premier sommet 5.



(b) Arbre des appels pour l'algorithme de Bron-Kerbosch *avec pivot* exécuté sur son premier sommet 5. Lorsque P contient plus d'un sommet, le pivot sélectionné est affiché en rouge. Selon la règle de Tomita *et al* [TTT06], le pivot est choisi de telle manière qu'il partage le plus de sommets possibles avec P .

FIGURE 4.2 – Exemple des appels récursifs de l'algorithme de Bron-Kerbosch avec et sans pivot, sur le graphe de la figure 4.1. Le calcul correspond au traitement du sommet 5 en supposant que ce soit le premier traité dans l'ordre des sommets. Pour chaque appel, les contenus des ensembles R , P et X sont donnés tels qu'ils sont en argument de l'appel. Les feuilles de l'arbre en vert correspondent aux cliques maximales renvoyées, et les autres sont des feuilles qui ne renvoient pas de clique maximale.

Algorithme 4.3: Algorithme d’Eppstein *et al.* [ELS10].

Entrée: Graphe $G = (V, E)$.

Sortie: Ensemble des cliques maximales de G .

```

1 for each  $v_i$  triés par dégénérescence  $v_0, v_1, \dots, v_{n-1}$  dans  $V$  do
2    $P \leftarrow N(v_i) \setminus \{v_0, \dots, v_{i-1}\}$ 
3    $X \leftarrow N(v_i) \cap \{v_0, \dots, v_{i-1}\}$ 
4   BKPivot( $\{v_i\}, P, X$ )

```

Dans leurs travaux, Eppstein *et al.* montrent que la complexité de l’appel initial $\text{BKPivot}(\{v_i\}, P, X)$ de la ligne 4 est exponentielle en $|P|$. Néanmoins, Eppstein *et al.* ont montré qu’en ordonnant les sommets selon leur dégénérescence, l’ensemble P construit pour un sommet v_i est de taille au plus la dégénérescence de v_i , et non pas son degré. Un exemple de la dégénérescence d’un sommet peut être donné dans le graphe de la figure 4.1 : la dégénérescence du sommet 4 vaut 3, tandis que son degré est 5. Ainsi, en tirant profit de cet ordre, ils montrent que la complexité de l’algorithme devient en $\mathcal{O}(n \cdot c \cdot 3^{c/3})$, où c est la dégénérescence du graphe, ce qui réduit considérablement le facteur exponentiel dans le cas des graphes du monde réel, qui peuvent avoir un degré maximal élevé du fait de leur caractère hétérogène, mais ont une en général une faible dégénérescence. Cette complexité est alors bien meilleure que celle donnée par Tomita *et al.* [TTT06] dans notre cas d’intérêt.

Dans leur article, Eppstein *et al.* décrivent une structure de données capable de mettre en œuvre très efficacement cette méthode sur de gros jeux de données massifs issus du monde réel. Ils en fournissent une implémentation C++ qui est une référence d’efficacité d’énumération des cliques maximales et qui est utilisée de manière systématique comme état de l’art dans les études expérimentales récentes².

4.2.2 Complexité en fonction de la sortie

Nous formulons ici la complexité de Eppstein *et al.*, dans les cas avec et sans pivot. L’algorithme sans pivot consiste à utiliser la fonction BK à la place de la fonction BKPivot à la ligne 4 de l’algorithme 4.3. Nous donnons cette complexité comme une fonction de :

- α : le nombre de cliques maximales de G ;
- q : la taille d’une plus grande clique ;
- c : la dégénérescence de G ;
- d : le degré maximal de G .

Pour ce faire, nous considérons les arbres des appels récursifs pour chacun de ces algorithmes. Les nœuds internes de ces arbres correspondent aux appels pour

2. <https://github.com/darrenstrash/quick-cliques>

lesquels l'ensemble de sommets parcouru par la boucle qui lance les appels récursifs n'est pas vide (ligne 4 de l'algorithme 4.1 et ligne 5 de l'algorithme 4.2), c'est-à-dire qu'ils génèrent d'autres appels enfants tandis que les feuilles correspondent aux appels qui n'en génèrent aucun autre.

Inspirés par les travaux de Conte *et al.* [CT22], nous nous concentrons dans ce qui suit sur les feuilles de ces arbres des appels, que nous séparons en deux catégories : celles qui renvoient une clique maximale et celles qui n'en renvoient pas. Ces dernières correspondent à des calculs inutiles, car elles ne contribuent pas à l'énumération. Une stratégie optimale de pivot couperait ces branches, de sorte à ne laisser que des feuilles qui renvoient une clique maximale. Notons ℓ le nombre total de feuilles des arbres des appels. Certaines de ces feuilles sont des cliques maximales (il y en a α), et d'autres ne le sont pas. Nous nous intéressons alors au rapport de “bonnes” feuilles :

$$r = \frac{\alpha}{\ell}.$$

Ce rapport peut être calculé soit pour l'algorithme 4.1 (sans pivot), soit pour l'algorithme 4.2 (avec pivot). Dans les deux cas, il quantifie la proportion $1 - r$ de branches qui resteraient à élaguer pour une énumération optimale. En particulier, si r est inférieur à 1, cela signifie qu'il y a des appels récursifs qui ne sont pas nécessaires. Sur un même graphe, la comparaison des rapports obtenus par les deux algorithmes avec et sans pivot montre dans quelle mesure la stratégie du pivot a été efficace pour élaguer un grand nombre d'appels inutiles ou non. Par exemple, dans l'arbre des appels donné dans la figure 4.2, il y a deux cliques maximales énumérées et l'algorithme avec pivot génère 5 feuilles, donc le rapport est de $\frac{2}{5}$, tandis que l'algorithme avec pivot a un rapport de 1 (mais il manque les appels sur les autres sommets pour avoir la valeur de r sur l'ensemble de l'algorithme).

L'utilisation de ce rapport nous permet d'établir le théorème 4.1 pour décrire la complexité de l'algorithme 4.3 en fonction de sa sortie.

Théorème 4.1 (Complexité en fonction de la sortie). *Avec la définition de r donnée ci-dessus, on a $1 \leq \frac{1}{r} \leq 2^q$, et la complexité de l'algorithme 4.3 est en $\mathcal{O}\left(\frac{1}{r} \cdot c \cdot d \cdot q \cdot \alpha\right)$. En remplaçant les appels à BKPivot par des appels à BK, l'expression de la complexité est la même, avec un facteur $\frac{1}{r}$ différent.*

Démonstration. Montrons d'abord que $1 \leq \frac{1}{r} \leq 2^q$. D'une part, il est clair que $\frac{1}{r} \geq 1$. D'autre part, chaque clique maximale contient au plus 2^q sous-cliques, donc il y a au plus $2^q \cdot \alpha$ cliques au total dans le graphe (maximales ou non). Or Conte *et al.* [CT22] ont montré que chaque nœud des arbres des appels à BKPivot correspond à une clique R différente des autres. Ainsi, comme chaque feuille est en particulier un nœud d'un arbre des appels, on en déduit que $\ell \leq 2^q \cdot \alpha$, et donc $\frac{1}{r} \leq 2^q$.

Exprimons la complexité. Par définition de q , nous savons que la profondeur des arbres des appels est au plus q . Il y a donc au plus $q \cdot \ell$ nœuds. Or les opérations au sein d'un appel se font en $\mathcal{O}(c \cdot d)$. En effet, comme justifié par Eppstein *et al.* en utilisant un ordre par dégénérescence, $|P|$ vaut au plus c [ELS10]. Le pivot est donc choisi en $\mathcal{O}(d \cdot c)$, en testant la taille de $P \cap N(p)$ pour chaque $p \in P \cup X$. De plus, les intersections $P \cap N(u)$ et $X \cap N(u)$ de la ligne 6 de **BKPivot** se font en $\mathcal{O}(d)$, et la boucle de la ligne 4 itère sur au plus $|P|$ sommets, donc elle s'exécute elle aussi en $\mathcal{O}(c \cdot d)$ (sans compter les opérations au sein des appels récursifs enfants). Donc la complexité des algorithmes 4.1 et 4.2 est en $\mathcal{O}(c \cdot d \cdot q \cdot \ell)$. Comme $\ell = \frac{1}{r} \cdot \alpha$, alors la complexité est en $\mathcal{O}\left(c \cdot d \cdot q \cdot \frac{1}{r} \cdot \alpha\right)$.

Notez que la complexité est la même lorsqu'on utilise la fonction **BK** à la place de **BKPivot**. La seule différence dans la preuve provient du calcul du pivot, qui n'est pas réalisé par la fonction **BK**. \square

Il est à noter que les bornes pour $\frac{1}{r}$ données par le théorème 4.1 sont précises. Premièrement, $\frac{1}{r}$ peut être égal à 1, par exemple dans le cas d'une seule clique, traitée dans l'algorithme avec pivot, pour lequel il n'y a qu'une seule branche, et donc une seule feuille. Cependant, $\frac{1}{r}$ peut également être exponentiel en q , dans le cas de l'algorithme sans pivot. Considérons le même graphe formé d'une seule clique, mais traité dans l'algorithme de Bron-Kerbosch sans pivot. Il ne contient alors qu'une seule clique maximale, de sorte que $\frac{1}{r} = \ell$. Il est possible de montrer qu'il y a $\ell = 2^{q-2}$ feuilles dans l'ensemble des arbres des appels³. C'est le cas par exemple de la clique $\{2, 3, 4, 5\}$ dans l'arbre des appels de la figure 4.2 (a). Si on supprime le nœud 7, cet arbre contiendrait 4 feuilles, qui est bien égal à 2^{4-2} . Ainsi, l'algorithme 4.1 sur un tel graphe atteint $\frac{1}{r} = 2^{q-2}$.

Néanmoins, Conte *et al.* [CT22] montrent que r est proche de 1 en pratique dans le cas de l'algorithme **BKPivot**, ce qui fait que ce rapport $\frac{1}{r}$ reste faible. D'après cette observation et la complexité calculée, nous pouvons affirmer que le nombre d'opérations effectuées par l'algorithme avec pivot est proche du meilleur que l'on puisse attendre. En effet, puisque l'algorithme doit traiter tous les sommets de toutes les α cliques maximales, et que la taille des plus grandes cliques est de q , alors on s'attend à ce qu'il y ait nécessairement un facteur $q \cdot \alpha$ dans la complexité. L'algorithme d'Eppstein *et al.* n'y ajoute qu'un facteur multiplicatif $\frac{1}{r} \cdot c \cdot d$, ce qui explique ses bonnes performances sur les graphes issus du monde réel, pour lesquels la plupart des dégénérescences sont faibles en pratique.

3. En effet, si on note $\ell(q)$ le nombre de feuilles des arbres des appels pour l'algorithme utilisant Bron-Kerbosch sans pivot sur une clique de taille q , alors en traitant les q sommets un par un, on obtient que $\ell(q) = \sum_{i=0}^{q-1} \ell(i)$, avec $\ell(1) = 1$ et $\ell(0) = 0$. Donc $\ell(q) = 2^{q-2}$.

4.2.3 Travaux récents sur l'énumération des cliques maximales

Depuis la publication des travaux de Eppstein *et al.* [ELS10], plus d'une dizaine de travaux ont été réalisés, mais la plupart sont des adaptations directes de l'algorithme de Bron-Kerbosch. On peut différencier ces travaux en trois catégories.

Tout d'abord, la plupart des travaux de recherche dans ce domaine se sont concentrés sur la parallélisation de l'algorithme d'énumération de cliques maximales [DDZ14 ; Les+17 ; YL19 ; Bla+20 ; DST18 ; WCT21]. Le constat de départ étant que le facteur limitant de l'énumération est son temps d'exécution, ces méthodes permettent de l'accélérer significativement, bien qu'elles ne proposent pas d'amélioration de l'algorithme séquentiel.

De nouveaux algorithmes ont été proposés pour la sélection du pivot, qui améliorent dans certains cas le temps de l'énumération, mais sans changer son ordre de grandeur. En 2016, Naudé [Nau16] a proposé un algorithme plus efficace pour calculer le pivot de Tomita, qui ne requiert pas de tester tous les sommets de P . En 2022, Jin *et al.* ont proposé un pivot heuristique, qui n'est pas le meilleur dans le pire des cas, contrairement à celui de Tomita, mais qui est plus rapide à calculer et donc améliore le temps de calcul de l'énumération [Jin+22]. De plus, ils ont présenté une méthode d'implémentation efficace qui part de celle proposée par Eppstein *et al.*, en l'améliorant grâce à des petites matrices d'adjacence partielles au sein des appels récursifs, pour accélérer les tests d'adjacence. Sur certains graphes, leur méthode va jusqu'à diviser par deux le temps de calcul par rapport à l'implémentation de Eppstein *et al.*.

Enfin, certains travaux améliorent la compréhension de la complexité théorique du problème. Dans ce domaine, les travaux de Conte *et al.* [Con+20] en 2020 et de Conte et Tomita en 2022 [CT22] ont été particulièrement significatifs. En 2020, ils ont élaboré un algorithme qui permet d'avoir une borne polynomiale sur le délai de calcul entre deux cliques maximales. De plus, ils ont montré que le délai entre deux cliques énumérées par l'algorithme de Bron-Kerbosch peut être exponentiel, et donc que leur approche est plus intéressante de ce point de vue. Ils ont également montré qu'il a une borne pour la mémoire qui est sous-linéaire. Mais bien que la complexité de leur algorithme soit meilleure, celui d'Eppstein *et al.* reste en moyenne 3,7 fois plus rapide. En revanche, ils ont réussi à avoir une meilleure consommation mémoire grâce à leur méthode sous-linéaire, tandis que la mémoire est linéaire dans l'algorithme d'Eppstein *et al.* Néanmoins, pour l'énumération des cliques maximales, c'est le temps de calcul qui est limitant et non mémoire. Enfin, en 2022, Conte et Tomita ont proposé la complexité de l'algorithme de Bron-Kerbosch en fonction de la sortie : $\mathcal{O}(\alpha \cdot 2^q \cdot n^2 \cdot d)$. Il s'agit de la première formule de complexité qui permet d'analyser l'algorithme directement en fonction de son nombre de cliques maximales, et nous nous en sommes inspiré pour développer le théorème 4.1.

4.3 État de l'art de l'énumération des bicliques maximales

Dans tout ce qui suit, nous fixons un graphe biparti $G = (V, U, E)$. Pour rappel, une biclique est une paire d'ensembles de sommets (C_U, C_V) avec $C_U \subseteq U$ et $C_V \subseteq V$ tels que les sommets de C_U sont tous connectés aux sommets de C_V . Un exemple est donné dans la figure 4.3.

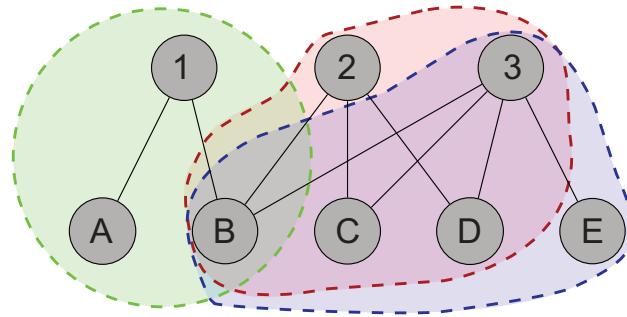


FIGURE 4.3 – Exemple d'un graphe biparti, et trois bicliques maximales entourées en couleur : $(\{1\}, \{A, B\})$, $(\{2, 3\}, \{B, C, D\})$ et $(\{3\}, \{B, C, D, E\})$.

Dans la littérature, de nombreux travaux ont été consacrés à l'énumération des bicliques maximales [UKA+04 ; Li+07 ; Dam14 ; Zha+14 ; DST18 ; Abi+20 ; Che+22]. Le plus récent est celui de Chen *et al.* [Che+22], qui ont publié en 2022 un algorithme qui est une amélioration directe du travail publié par Abidi *et al.* [Abi+20], et qui est beaucoup plus performant que tous les autres algorithmes de l'état de l'art.

Cet algorithme s'inspire de celui de Bron-Kerbosch, mais il ne suit pas la logique qui consiste à compléter les bicliques en ajoutant tous ses sommets un par un. Il fait grossir la biclique en ajoutant les sommets un par un uniquement ***d'un seul côté*** du graphe biparti, l'autre côté de la biclique étant déterminé directement comme l'intersection des voisins des sommets sélectionnés.

De plus, les auteurs semblent s'inspirer des travaux d'Eppstein *et al.* décrits plus haut : ils considèrent l'un des deux ensembles de sommets du graphe biparti, que l'on note U dans ce paragraphe, et ils énumèrent les bicliques à partir de chaque sommet de U , selon un ordre donné, en suivant le même principe que celui décrit dans l'algorithme 4.3, mais adapté au cas biparti. L'ordre qu'ils considèrent est un ordre de dégénérescence du graphe projeté sur U , qui est le graphe défini par le voisinage projeté N_2 , où les voisins d'un sommet u par N_2 sont les sommets de U qui ont au moins un voisin en commun avec u dans le graphe biparti (voir section 2.1 préliminaire). Ils montrent que la complexité de cet algorithme est en $\mathcal{O}(n_U \cdot \zeta_U \cdot m \cdot 2^{\zeta_U})$, où n_U est le nombre de sommets dans U , ζ_U est la dégénérescence du graphe projeté sur U et m est le nombre d'arêtes dans G . Ils expriment également la complexité

de leur algorithme en termes de nombre de bicliques maximales \mathcal{B} : $\mathcal{O}(\zeta_U \cdot m \cdot \mathcal{B})$. Notons que même pour un graphe peu dense, le graphe projeté sur U peut être très dense, et donc avoir une valeur élevée de ζ_U .

Leur méthode est très efficace, mais elle présente le défaut que leur pivot se calcule en temps cubique, alors que celui de Bron-Kerbosch se calcule en temps quadratique. Cela laisse penser qu'une adaptation plus directe de l'algorithme de Bron-Kerbosch est à mettre en œuvre, et c'est ce que nous proposons dans ce chapitre.

Dans tout ce chapitre, nous prenons comme référence les travaux de Chen *et al.* [Che+22], de loin l'algorithme qui fournit les meilleurs résultats en termes de temps de calcul. Cet algorithme s'appelle OOMBEA. Nous nous en servons comme point de comparaison de notre nouvel algorithme d'énumération des bicliques maximales.

4.4 Un nouvel algorithme d'énumération des bicliques maximales

Dans cette section, nous montrons que l'algorithme de Eppstein *et al.*, qui est le meilleur algorithme d'énumération sur les graphes issus du monde réel et que nous avons décrit dans la section 4.2, peut être directement appliqué aux graphes bipartis, afin d'obtenir l'ensemble des bicliques maximales. Cela nous permet d'obtenir un algorithme simple pour l'énumération des bicliques maximales.

Dans tout ce qui suit, nous fixons un graphe biparti $G = (U, V, E)$. Si A est un ensemble de sommets de G , alors on note \mathbf{A}_U l'ensemble des sommets de A qui sont dans U , c'est-à-dire $A \cap U$, et \mathbf{A}_V l'ensemble des sommets de A qui sont dans V : $A \cap V$. Nous rappelons la définition du **voisinage projeté** d'un sommet $x \in U \cup V$, que nous avons donné dans la section 2.1.2 préliminaire, noté $N_2(x)$ et qui correspond à l'ensemble des voisins des voisins de x :

$$N_2(x) = \{u \in U \mid \exists v \in N(x), u \in N(v)\} \text{ si } x \in U,$$

$$N_2(x) = \{v \in V \mid \exists u \in N(x), v \in N(u)\} \text{ si } x \in V.$$

Par exemple, dans le graphe biparti de la figure 4.3, $N_2(1) = N_2(2) = N_2(3) = \{1, 2, 3\}$, $N_2(A) = \{A, B\}$, $N_2(B) = \{A, B, C, D, E\}$ et $N_2(C) = N_2(D) = N_2(E) = \{B, C, D, E\}$.

Le **graphe projeté sur U** (resp. sur V) est le graphe $G_2(U) = (U, E_2(U))$ défini par le voisinage N_2 , **duquel on retire les boucles** : $E_2(U) = \{\{u_1, u_2\} \mid u_1 \neq u_2 \text{ et } u_1 \in N_2(u_2)\}$.

4.4.1 Graphe étendu d'un graphe biparti

Pour étendre l'énumération des cliques maximales dans les graphes à l'énumération des bicliques maximales dans les graphes bipartis, nous définissons le graphe étendu d'un graphe biparti, qui correspond au graphe auquel on ajoute des arêtes entre tous les sommets de U , et entre tous les sommets de V . Nous notons ce graphe \tilde{G} et nous le définissons formellement ci-dessous.

Définition 4.1 (Graphe étendu d'un graphe biparti). *Le **graphe étendu du graphe biparti G** est le graphe $\tilde{G} = (U \cup V, \tilde{E})$, où :*

$$\tilde{E} = E \cup \{\{u_1, u_2\} \in U^2 \mid u_1 \neq u_2\} \cup \{\{v_1, v_2\} \in V^2 \mid v_1 \neq v_2\}.$$

Ce graphe étendu induit une notion de voisinage, que l'on appelle voisinage étendu, noté \tilde{N} .

Définition 4.2 (Voisinage étendu d'un sommet). *Soit $x \in U \cup V$. Le **voisinage étendu de x** correspond aux voisins de x dans le graphe étendu. On le note $\tilde{N}(x)$:*

$$\tilde{N}(x) = N(x) \cup U \setminus \{x\} \text{ si } x \in U,$$

$$\tilde{N}(x) = N(x) \cup V \setminus \{x\} \text{ si } x \in V.$$

Ce graphe a une propriété particulière, que nous allons exploiter dans notre algorithme d'énumération des bicliques maximales : un ensemble de sommets qui forme une clique de \tilde{G} forme de manière équivalente une biclique de G . Ce résultat a été introduit par Gély *et al.* [GNS09]. Nous le formalisons dans le théorème ci-dessous.

Théorème 4.2. *Soit $G = (U, V, E)$ un graphe biparti. Alors, les cliques maximales de \tilde{G} sont les bicliques maximales de G :*

$$C \text{ est une clique maximale de } \tilde{G} \Leftrightarrow (C_U, C_V) \text{ est une biclique maximale de } G.$$

Démonstration. Tout d'abord, remarquons que si C est une clique de \tilde{G} , alors par définition, $\forall u \in C_U, \forall v \in C_V, v \in \tilde{N}(u)$, et donc $v \in N(u)$, ce qui implique que (C_U, C_V) est une biclique de G . Réciproquement, soit (C_U, C_V) une biclique de G . Alors, $\forall u \in C_U, \forall v \in C_V, v \in N(u)$, donc $v \in \tilde{N}(u)$. De plus, $\forall u_1, u_2 \in C_U$ tels que $u_1 \neq u_2, u_1 \in \tilde{N}(u_2)$ par construction de \tilde{N} . De même, chaque paire de sommets de C_V est connectée par \tilde{N} , et donc C est une clique de \tilde{G} .

Ainsi, on obtient le premier résultat suivant :

$$C \text{ est une clique de } \tilde{G} \Leftrightarrow (C_U, C_V) \text{ est une biclique de } G.$$

Maintenant, fixons C une clique de \tilde{G} et (C_U, C_V) la biclique associée dans G , et montrons que C est maximale si et seulement si (C_U, C_V) est maximale. Pour cela, on établit l'équivalence des négations :

$$\begin{aligned} & C \text{ n'est pas maximale} \\ \Leftrightarrow & \text{il existe } x \in (U \cup V) \setminus C \text{ tel que } C \cup \{x\} \text{ est une clique de } \tilde{G} \\ \Leftrightarrow & \text{il existe } x \in (U \cup V) \setminus (C_U \cup C_V) \text{ tel que } (C_U, C_V) \cup \{x\} \text{ est une biclique de } G \\ \Leftrightarrow & (C_U, C_V) \text{ n'est pas maximale.} \end{aligned}$$

□

Ce théorème induit une méthode directe d'énumération des bicliques maximales d'un graphe biparti : il suffit d'énumérer les cliques maximales de son graphe étendu. L'algorithme 4.4 propose le pseudocode pour cette méthode : il s'agit simplement de l'algorithme de Eppstein *et al.* [ELS10], dans lequel nous remplaçons N par \tilde{N} , et le théorème 4.2 en montre directement la validité.

Algorithme 4.4: Énumérer les bicliques maximales avec le graphe étendu.

Entrée: Graphe biparti $G = (U, V, E)$.

Sortie: Ensemble des bicliques maximales de G .

```

1 for each  $x_i$  selon un ordre  $x_0, x_1, \dots, x_{n-1}$  de  $U \cup V$  do
2    $P \leftarrow \tilde{N}(x_i) \setminus \{x_0, \dots, x_{i-1}\}$ 
3    $X \leftarrow \tilde{N}(x_i) \cap \{x_0, \dots, x_{i-1}\}$ 
4   BK-Étendu( $\{x_i\}, P, X$ )
5 Fonction BK-Étendu( $R, P, X$ ):
6   if  $P \cup X = \emptyset$  then
7     return  $R$  biclique maximale
8    $p \leftarrow$  pivot dans  $P \cup X$ 
9   for  $x \in P \setminus \tilde{N}(p)$  do
10     $\quad$  BK-Étendu( $R \cup \{x\}, P \cap \tilde{N}(x), X \cap \tilde{N}(x)$ )
11     $P \leftarrow P \setminus \{x\}$ 
12     $X \leftarrow X \cup \{x\}$ 
```

Néanmoins, cet algorithme n'est pas utilisable en pratique pour des graphes contenant beaucoup de sommets, car les ensembles de recherche P définis à la ligne 2 sont de taille au moins $|U|^2$. Cela nous amène au développement de notre nouvel algorithme, BBK.

4.4.2 BBK : nouvel algorithme d'énumération des bicliques maximales

Ensembles de sommets sur lesquels faire l'énumération. Pour exploiter l'algorithme 4.4 dans le cas des graphes bipartis massifs du monde réel, nous dressons plusieurs constats. Premièrement, il n'est pas nécessaire d'énumérer pour chaque sommet $x \in U \cup V$ l'ensemble des bicliques maximales qui contiennent x . À la place, on peut se contenter de n'énumérer que les bicliques maximales qui contiennent un sommet $u \in U$. En effet, (\emptyset, V) est une biclique, donc c'est la seule biclique maximale éventuelle qui ne contient aucun sommet de U . Elle peut donc être renvoyée en dehors de l'énumération.

Deuxièmement, pour calculer les cliques maximales dans un graphe, l'algorithme de Bron-Kerbosch utilise le fait que les sommets qui appartiennent à une clique avec un sommet u , sont les voisins de ce sommet u . Cela permet de déterminer le voisinage d'une clique comme l'intersection des voisainages des sommets qui la composent, et c'est cette propriété qui valide l'algorithme de Bron-Kerbosch. Dans un graphe biparti, ce n'est pas le cas : si $u \in U$, l'ensemble des sommets qui sont dans une biclique avec u sont en fait **les sommets de $N(u) \cup N_2(u) \setminus \{u\}$** ⁴. Ce résultat est démontré par Hermelin et Manoussakis [HM21], et nous nous en servons comme base pour le développement de notre algorithme : pour chaque sommet $u \in U$, il suffit de chercher les bicliques qui contiennent u parmi les sommets de l'ensemble $N(u) \cup N_2(u) \setminus \{u\}$. Cet ensemble est de taille au plus $d_2(u) + d(u) + 1$.

Enfin, comme on l'a vu, Eppstein *et al.* ont montré que l'ordre des sommets avait un impact sur l'efficacité de l'énumération : ils utilisent un ordre par dégénérescence des sommets afin de réduire la taille des ensembles de sommets sur lesquels rechercher les cliques. De la même manière, nous associons une dégénérescence à cette valeur sur U , que nous appelons la **bidégénérescence** sur U . Nous choisissons d'ordonner les sommets de U en suivant cette bidégénérescence, pour que les grandes valeurs de $d_2(u) + d(u) - 1$ soient traitées en dernier, afin de réduire la taille des ensembles de recherche. Nous donnons une définition formelle de ces notions ci-dessous.

Définition 4.3 (Degré augmenté d'un sous-graphe H de $G_2(U)$). *Soit $u \in U$. Soit H un sous-graphe du graphe projeté sur U , $G_2(U)$, qui contient u . On appelle le **degré augmenté** de u dans H la valeur $d_{H+}(u) = |N_2(u) \cap H| + d(u) - 1$, où d est le degré dans le graphe biparti G .*

Définition 4.4 (Bidégénérescence d'un sommet). *La **bidégénérescence** d'un sommet $u \in U$, que l'on note $b_U(u)$, est la dégénérescence de u dans $G_2(U)$, définie pour le degré augmenté : $b_U(u)$ est la valeur maximale telle qu'il existe un sous-graphe induit $H = (V_H, E_H)$ de $G_2(U)$ contenant u et vérifiant : $\forall x \in V_H, d_{H+}(x) \geq b_U(u)$.*

4. Notez que cela ignore la biclique maximale éventuelle (U, \emptyset) , qu'il faut alors énumérer par ailleurs.

Par exemple, dans la figure 4.3, la bidégénérescence de A vaut 2, et celle de B , C , D et E vaut 4.

Définition 4.5 (Bidégénérescence de U et de V). *La **bidégénérescence de U** (resp. V), notée b_U (resp. b_V), est la bidégénérescence maximale des sommets de U (resp. V).*

À partir de ces notions, nous définissons un ordre par bidégénérescence de la même manière qu'un ordre par dégénérescence sur les graphes :

Définition 4.6 (Ordre des sommets de U par bidégénérescence). *Nous appelons un **ordre par bidégénérescence de U** (resp. V), une liste des sommets de U (resp. V) obtenue en sélectionnant itérativement le sommet de plus petit degré augmenté sur $G_2(U)$ et en diminuant de 1 le degré augmenté de tous ses voisins.*

Cet ordre par bidégénérescence se calcule comme un ordre par dégénérescence du graphe $G_2(U)$, en temps linéaire sur la taille de ce graphe.

De la même manière que la dégénérescence dans les graphes, cet ordre u_1, \dots, u_n sur les sommets de U permet de réduire la taille maximale des ensembles $(N_2(u_i) \cup N(u_i) \setminus \{u_i\}) \setminus \{u_0, \dots, u_{i-1}\}$, et c'est ce que nous utilisons pour notre algorithme. $(N_2(u_i) \cup N(u_i) \setminus \{u_i\}) \setminus \{u_0, \dots, u_{i-1}\}$ est au plus de taille b_U .

Pseudocode de l'algorithme. Toutes ces considérations nous mènent au développement de notre algorithme d'énumération des bicliques maximales, que l'on appelle BBK, et dont le pseudocode est donné dans l'algorithme 4.5, où on affiche en rouge les modifications apportées par rapport à l'algorithme 4.4.

Algorithme 4.5: BBK : énumération des bicliques maximales.

Entrée: Graphe biparti $G = (U, V, E)$.

Sortie: Ensemble des bicliques maximales de G .

```

1 for each  $u_i$  selon un ordre par bidénérescence  $u_0, u_1, \dots, u_{n-1}$  de  $U$  do
2    $P \leftarrow (N(u_i) \cup N_2(u_i) \setminus \{u_i\}) \setminus \{u_0, \dots, u_{i-1}\}$ 
3    $X \leftarrow (N(u_i) \cup N_2(u_i) \setminus \{u_i\}) \cap \{u_0, \dots, u_{i-1}\}$ 
4   BK-Étendu( $\{u_i\}, P, X$ )

```

Tirer profit du caractère clairsemé du graphe biparti : utiliser N à la place de \tilde{N} . Au premier abord, la fonction BK-Étendu utilisée dans notre algorithme BBK et définie dans l'algorithme 4.4 ne semble pas efficace. En effet, les voisinages \tilde{N} sont trop gros pour être traités efficacement. En fait, toute la stratégie de cet algorithme consiste à ne pas avoir besoin d'utiliser ce \tilde{N} , mais seulement le voisinage N du graphe biparti.

L'ensemble $P \cap \tilde{N}(x)$ qui est calculé à la ligne 10 de l'algorithme 4.4 est égal à $P \cap (N(x) \cup U \setminus \{x\})$. Ainsi, lorsque $x \in U$ (resp. $x \in V$), cet ensemble est égal à $(P_U \setminus \{x\}) \cup (P_V \cap N(x))$ (resp. $(P_V \setminus \{x\}) \cup (P_U \cap N(x))$)⁵. Calculer cet ensemble n'utilise alors que le voisinage N du graphe biparti. Il en est de même pour l'opération $X \cap \tilde{N}(v)$, en suivant le même raisonnement.

Nous pouvons appliquer un raisonnement similaire à l'élagage du pivot de la ligne 9. En effet, l'ensemble $P \setminus \tilde{N}(p)$ est égal à $P \setminus (N(p) \cup U \setminus \{p\})$. Prenons le cas où $p \in U$: cet ensemble est égal à $(P_U \setminus (U \setminus \{p\})) \cup (P_V \setminus N(p))$, qui est égal à $\{p\} \cup (P_V \setminus N(p))$. Le raisonnement est similaire pour $p \in V$, où l'ensemble est égal à $\{p\} \cup (P_U \setminus N(p))$. Ainsi, l'opération d'élagage $P \setminus \tilde{N}(p)$ réalisée à la ligne 9 consiste à garder p , et à retirer ses voisins du côté de P auquel il n'appartient pas. On utilise ainsi uniquement le voisinage $N(p)$ pour calculer cet élagage, et non $N_2(p)$. Notez en particulier que le pivot permet d'élaguer tous les sommets (sauf lui-même) du côté où il est choisi, en plus de retirer les sommets de $N(p)$. Ceci est très intéressant en terme d'élagage des appels inutiles, par rapport à uniquement élaguer les sommets de $N(p)$.

Comme pour l'énumération des cliques maximales, le pivot est ici choisi de manière à maximiser le nombre de sommets élagués, c'est-à-dire à minimiser $P \setminus \tilde{N}(p)$.

Amélioration du test de maximalité des bicliques. Le test de maximalité de la ligne 6 peut être amélioré en tenant compte du caractère biparti du graphe. Si $P_U = \emptyset$, on peut tester deux cas en temps constants qui impliquent qu'il n'y a besoin de faire d'appel récursif sur aucun sommet de P_V :

- si $P_U = \emptyset$ et $X_V \neq \emptyset$, alors X_V ne sera plus jamais modifié dans les sous-appels récursifs de cette branche, car X_V n'est modifié que lorsqu'on ajoute un sommet de P_U à la biclique en construction. Donc on a $X_V \neq \emptyset$ pour tous les sous-appels et donc aucun ne peut correspondre à une biclique maximale. Ainsi, on peut stopper l'appel et ne faire aucun appel récursif sur les sommets de P_V ;
- si $P_U = \emptyset$ et $X_V = \emptyset$, alors si $X_U = \emptyset$, on peut remarquer que $(R_U, R_V \cup P_V)$ est une biclique maximale, et on peut la renvoyer directement, sans faire les appels récursifs.

Ces observations sont également à prendre en compte en échangeant les rôles de U et de V .

Dans la suite de ce chapitre, et notamment dans la section où nous expérimentons l'algorithme, ces améliorations sont prises en compte.

5. Nous rappelons que si $A \subseteq U \cup V$, la notation A_U (resp. A_V) correspond à l'ensemble $A \cap U$ (resp. $A \cap V$).

4.4.3 Complexité de l'algorithme

Comme pour l'énumération des cliques maximales, nous sommes en mesure de fournir la complexité de l'algorithme 4.5 BBK en fonction de l'entrée et de la sortie. Nous donnons ces complexités respectivement dans le théorème 4.3 et le théorème 4.4, après avoir introduit le lemme 4.1 suivant.

Lemme 4.1. *La bidégénérescence de U est plus grande que les degrés maximaux de U et de V :*

$$d_U \leq b_U \text{ et } d_V \leq b_U.$$

Démonstration. Soit $u \in U$ de degré d_U . Alors, le sous-graphe induit par $\{u\}$ sur $G_2(U)$ est tel que tous ses sommets ont un degré augmenté d'au moins d_U . On en déduit que $b_U \geq d_U$.

Considérons maintenant $v \in V$ de degré d_V . Alors, le sous-graphe induit par $N(v)$ sur $G_2(U)$ est une clique, donc ses sommets ont un degré augmenté d'au moins $d_V - 1$, auquel on ajoute 1, car leur voisinage dans G contient au moins v . Ainsi, de même, $b_U \geq d_V$.

□

Théorème 4.3. *La complexité de l'algorithme 4.5 BBK est en $\mathcal{O}(n_U \cdot b_U \cdot 3^{b_U/3})$, où n_U est le nombre de sommets de U et b_U la bidégénérescence du graphe sur U .*

Démonstration. L'algorithme commence par calculer un ordre par bidégénérescence. Pour chaque sommet $u \in U$, il calcule le voisinage projeté $N_2(u)$, en $\mathcal{O}(d_U \cdot d_V)$. Puis, l'ordre est calculé en prenant itérativement un sommet de degré augmenté minimal et en diminuant de 1 le degré augmenté de ses voisins dans le graphe projeté sur U . Toute cette procédure se réalise donc en $\mathcal{O}(n_U \cdot d_U \cdot d_V) \subseteq \mathcal{O}(n_U \cdot b_U^2)$ d'après le lemme 4.1.

Ensuite, pour chaque sommet $u_i \in U$ dans l'ordre calculé, l'algorithme énumère les bicliques maximales contenant u_i et aucun sommet le précédent dans l'ordre, à l'aide de la fonction **BK-Étendu** (ligne 4). Par définition de la bidégénérescence, on sait que la taille de l'ensemble $(N(u_i) \cup N_2(u_i) \setminus \{u_i\}) \setminus \{u_1, \dots, u_{i-1}\}$ est au plus de $b_U(u_i)$, donc au plus de b_U . Ainsi, on peut appliquer directement la formule de complexité de l'algorithme de Eppstein *et al.* [ELS10] que nous avons donnée dans la section 4.2.1 du chapitre 3, puisque les opérations réalisées sont les mêmes. Avec ces considérations, l'ensemble des appels à **BK-Étendu** se font donc en $\mathcal{O}(n_U \cdot b_U \cdot 3^{b_U/3})$.

L'algorithme s'exécute donc en $\mathcal{O}(n_U \cdot (b_U^2 + b_U \cdot 3^{b_U/3}))$. Or, on peut montrer que pour tout entier $k \geq 0$, $k^2 \leq k \cdot 3^{k/3}$. Ainsi, $b_U^2 \leq b_U \cdot 3^{b_U/3}$, et on en déduit la complexité recherchée.

□

Théorème 4.4. *La complexité de notre algorithme BBK en fonction de la sortie est en $\mathcal{O}\left(\frac{1}{r} \cdot b_U^2 \cdot q \cdot \mathcal{B}\right)$, où r est le rapport de Bron-Kerbosch défini dans la section 4.2.2, b_U la bidégénérescence sur U , q la taille maximale d'une biclique et \mathcal{B} le nombre de bicliques maximales.*

Démonstration. De la même manière que pour la preuve du théorème 4.3, l'algorithme BBK réalise les opérations de l'algorithme de Eppstein *et al.*, avec $|P|$ et les voisnages de taille au plus b_U . Donc, la complexité est la même que celle du théorème 4.1, en remplaçant d par b_U , d'où le résultat. □

Ces complexités sont à comparer à celles de l'algorithme OOMBEA de Chen *et al.*, qu'ils fournissent dans leur article. Ils montrent qu'il s'exécute en $\mathcal{O}(n_U \cdot \zeta_U \cdot m \cdot 2^{\zeta_U})$, où ζ_U est la dégénérescence du graphe projeté sur U et m est le nombre d'arêtes dans G . Les deux dégénérescences b_U et ζ_U n'étant pas définies de la même manière, il n'est pas évident de les comparer. Néanmoins, on constate que le facteur devant l'exponentielle est plus élevé pour leur algorithme, d'un facteur m , qui est élevé lorsque l'on traite des graphes massifs. D'autre part, OOMBEA s'exécute aussi en $\mathcal{O}(\zeta_U \cdot m \cdot \mathcal{B})$. En considérant notre facteur $\frac{1}{r}$ proche de 1, comme montré pour les cliques maximales dans les graphes, et que nous validons dans nos expériences, cela revient à comparer les deux facteurs $b_U^2 \cdot q$ et $\zeta_U \cdot m$.

4.5 Évaluation expérimentale

Dans cette section, nous réalisons des expériences sur notre algorithme BBK. Nous avons implémenté cet algorithme en C++, et le code est disponible en ligne⁶. Cette implémentation s'inspire de celle d'Eppstein *et al.* [ELS10]. Nous avons effectué nos expériences sur des machines équipées de 2 processeurs Intel Xeon E5645 avec 12 coeurs chacun et 64 Go de RAM. Lorsque ce n'est pas précisé, notre algorithme BBK est exécuté toujours en partant de l'ensemble de sommets U qui contient le moins de sommets dans le graphe biparti d'entrée $G = (U, V, E)$. Ce choix est discuté dans la section 4.5.3.

Nous commençons par présenter les graphes bipartis que nous utilisons dans ces expériences, puis nous comparons sur ces graphes les temps d'exécution de notre algorithme BBK à celui de l'algorithme OOMBEA de Chen *et al* [Che+22]. Ensuite, nous montrons l'influence du choix de l'ensemble U ou V à partir duquel initialiser les énumérations sur les sommets (ligne 1 de l'algorithme 4.5 BBK), puis nous discutons de l'impact de l'ordre sur le temps d'exécution. Enfin, nous montrons que bien que

6. <https://gitlab.lip6.fr/baudin/bbk>

notre implémentation soit meilleure en temps d'exécution, l'algorithme OOMBEA est beaucoup plus efficace en termes de mémoire.

4.5.1 Jeux de données

Nous avons testé notre algorithme BBK sur un ensemble de graphes bipartis récupéré sur la base de données KONECT [Kun13]. Nous avons sélectionné des graphes bipartis issus de situations réelles, correspondant à des nombres de sommets et d'arêtes variés et provenant de contextes divers, afin de tester notre algorithme dans différents scénarios.

Type de données. Les graphes que nous utilisons sont présentés dans le tableau 4.1. On peut séparer ces graphes en trois catégories. Une partie des graphes concernent des utilisateurs de plateformes en ligne : des tags postés dans *BibSonomy*⁷ et *CiteULike*⁸, les livres avec lesquels ils interagissent dans *BookCrossing*⁹, des notes déposées à des revues sur *DVD-Ciao*¹⁰, *FilmTrust*¹¹ et *WikiLens*¹², des posts réalisés sur des forums dans *UC-Forum*¹³. D'autres graphes relient des personnes à leurs travaux : *Actor-Movie*¹⁴ est un graphe reliant les acteurs aux films dans lesquels ils ont tourné, *CiteSeer*¹⁵ est un graphe qui relie des auteurs à leurs publications scientifiques, *GitHub*¹⁶ relie les utilisateurs aux projets sur lesquels ils travaillent. Enfin, les graphes restants classifient de l'information : *DailyKos*¹⁷ et *Reuters*¹⁸ connectent des documents et les mots qu'ils contiennent, *DBpedia*¹⁹ représente des athlètes et leur équipe, *TV-Tropes*²⁰ connecte des travaux artistiques à leur style, *Discogs*²¹ relie du contenu musical à son style, *Marvel*²² relie les personnages des Marvel aux publications dans lesquels ils apparaissent, *Pics*²³ connectent des personnes aux images sur lesquels ils sont tagués, et *YouTube*²⁴ connecte les utilisateurs aux groupes auxquels ils appartiennent.

-
7. <http://konect.cc/networks/bibsonomy-2ut>
 8. <http://konect.cc/networks/citeulike-ut>
 9. http://konect.cc/networks/bookcrossing_full-rating
 10. http://konect.cc/networks/librec-ciaodvd-review_ratings
 11. <http://konect.cc/networks/librec-filmtrust-ratings>
 12. <http://konect.cc/networks/wikilens-ratings>
 13. <http://konect.cc/networks/opsahl-ucforum>
 14. <http://konect.cc/networks/actor-movie>
 15. <http://konect.cc/networks/komarix-citeseer>
 16. <http://konect.cc/networks/github>
 17. <http://konect.cc/networks/bag-kos>
 18. <http://konect.cc/networks/gottron-reuters>
 19. <http://konect.cc/networks/dbpedia-team>
 20. <http://konect.cc/networks/dbtropes-feature>
 21. http://konect.cc/networks/discogs_lgenre
 22. <http://konect.cc/networks/marvel>
 23. http://konect.cc/networks/pics_ui
 24. <http://konect.cc/networks/youtube-groupmemberships>

Ces graphes sont triés de haut en bas dans le tableau 4.1 par nombre de bicliques maximales croissant (colonne \mathcal{B}). Les deux ensembles de sommets U et V de ces graphes bipartis $G = (U, V, E)$ sont choisis de telle manière que ce soit U qui contienne le moins de sommets ($n_U \leq n_V$). *DVD-Ciao* est le graphe contenant le plus de bicliques qu'on ait pu énumérer dans la limite d'une semaine de calcul. Certains graphes, plus massifs, ne nous ont pas permis d'obtenir l'ensemble des bicliques maximales dans cette limite de temps de calcul. Un premier constat intéressant à dresser est que le nombre de bicliques maximales ne semble pas du tout dépendre du nombre d'arêtes ni du nombre de sommets. Par exemple, le graphe *Actor-Movie* a quasiment autant d'arêtes que le graphe *DVD-Ciao*, et il a plus de sommets, mais il contient plus de cent fois moins de bicliques maximales.

Graphe	m	n_U	n_V	\mathcal{B}
<i>UC-Forum</i>	7 089	522	899	16 261
<i>Discogs</i>	481 661	15	270 771	17 650
<i>CiteSeer</i>	512 267	105 353	181 395	171 354
<i>Marvel</i>	96 662	6 486	12 942	206 135
<i>DBpedia</i>	1 366 466	34 461	901 130	517 943
<i>Actor-Movie</i>	1 470 404	127 823	383 640	1 075 444
<i>Pics</i>	997 840	17 122	495 402	1 242 718
<i>YouTube</i>	293 360	30 087	94 238	1 826 587
<i>WikiLens</i>	26 937	326	5 111	2 769 773
<i>BookCrossing</i>	1 149 739	105 278	340 523	54 458 953
<i>GitHub</i>	440 237	56 519	120 867	55 346 398
<i>DailyKos</i>	353 160	3 430	6 906	242 384 960
<i>FilmTrust</i>	35 494	1 508	2 071	646 318 495
<i>CiteULike</i>	538 761	22 715	153 277	2 333 281 521
<i>Reuters</i>	978 446	19 757	38 677	10 071 287 092
<i>BibSonomy</i>	453 987	5 794	204 673	10 526 275 315
<i>TV-Tropes</i>	3 232 134	64 415	87 678	19 636 996 096
<i>DVD-Ciao</i>	1 625 480	21 019	71 633	109 769 732 096

TABLEAU 4.1 – Jeux de données utilisés dans nos expériences, triés de haut en bas par nombre de bicliques maximales croissant. m représente le nombre de liens du graphe, n_U le nombre de sommets de son ensemble U qui en contient le moins, n_V le nombre de sommets de son ensemble V qui en contient le plus, et \mathcal{B} le nombre de bicliques maximales. .

Degrés et bidégénérescences des graphes. Le tableau 4.2 présente les degrés maximaux dans ces graphes bipartis, ainsi que leur bidégénérescence, définie dans la section 4.4.2, qui est utilisée dans l'algorithme BBK et qui apparaît dans sa complexité. Le tableau 4.2 montre, comme attendu, qu'il y a un écart entre le degré

maximal du graphe dans U ou V (\mathbf{d}_U ou \mathbf{d}_V) et le degré maximal du graphe projeté sur U ou sur V (\mathbf{d}_{2U} ou \mathbf{d}_{2V}). Le degré maximal des projetés est plus élevé, mais cet écart est limité dans la plupart des cas, souvent du même ordre de grandeur, et ne dépasse pas un facteur dix fois supérieur, à l'exception de *Pics* et *BookCrossing*. La bidégénérescence, quant à elle, est quasiment systématiquement égale dans U et dans V , et elle vaut le degré maximal du graphe biparti. Cette valeur est la meilleure que l'on puisse espérer car, comme on l'a vu dans le lemme 4.1, la bidégénérescence est au moins égale au degré maximal. Néanmoins, il y a un gain clair dans l'utilisation de la bidégénérescence, car le degré maximal du U est le maximum de $|N(u)|$ pour $u \in U$, et est donc plus faible que la valeur maximale de $|N(u) \cup N_2(u) \setminus \{u\}|$, qui pourrait être la taille maximale d'un ensemble de recherche P si les sommets n'étaient pas ordonnés par bidégénérescence dans l'algorithme 4.5 BBK. Néanmoins, ces valeurs sont des valeurs maximales et elles ne représentent pas les caractéristiques générales sur les sommets.

Degrés et bidégénérescences des sommets. Le tableau 4.3 présente les valeurs moyennes des degrés et de la bidégénérescence des sommets de chaque graphe biparti. On note une barre au-dessus des symboles ($\overline{\mathbf{d}_U}$, $\overline{\mathbf{d}_V}$, $\overline{\mathbf{d}_{2U}}$, $\overline{\mathbf{d}_{2V}}$, $\overline{\mathbf{b}_U}$ et $\overline{\mathbf{b}_V}$) pour annoncer que ce sont des valeurs moyennes. On observe que le comportement de ces valeurs est très différent de celui de leur valeur maximale. En effet, les degrés moyens sont faibles en moyenne par rapport à leurs valeurs maximales et la bidégénérescence moyenne d'un sommet est beaucoup plus élevée, plutôt proche du degré projeté moyen, lui aussi élevé. Les valeurs maximales ne permettent pas de rendre compte de cet écart. Ces valeurs sont intéressantes, car pour chaque sommet, l'énumération des bicliques contenant ce sommet est exécutée à partir d'un ensemble P de la taille de la bidégénérescence de ce sommet (ligne 2 de l'algorithme 4.5). Ainsi, cela donne un aperçu en moyenne de la taille de ces ensembles sur lesquels les bicliques sont énumérées, qui est élevé par rapport au degré moyen.

4.5.2 Résultats : temps de calcul

Pour mesurer le gain d'efficacité dans l'énumération des bicliques maximales apporté par notre algorithme, nous avons mesuré les temps de calcul de notre implémentation, ainsi que de celle fournie par Chen *et al.*, afin de les comparer²⁵. La figure 4.4 présente ces temps de calcul sous forme de barres et les valeurs exactes sont données dans le tableau 4.4. La figure 4.4 représente les temps de calcul des graphes triés en abscisse par nombre de bicliques maximales croissantes. Les barres en bleu correspondent aux temps de calcul de notre algorithme BBK, et les barres oranges à l'algorithme OOMBAA. L'échelle de temps est logarithmique, et on observe que le temps de calcul croît globalement avec le nombre de bicliques, alors qu'il ne semble pas directement relié par exemple au nombre de sommets, d'arêtes, au

25. https://github.com/S1mpleCod/cohesive_subgraph_bipartite

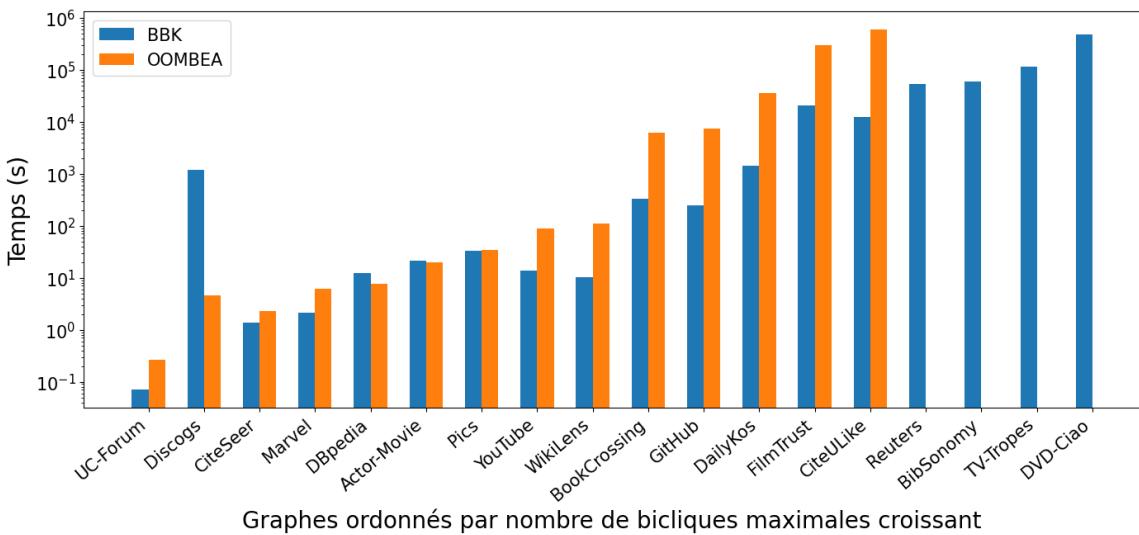
Graph	d_U	d_V	d_{2U}	d_{2V}	b_U	b_V
<i>UC-Forum</i>	126	99	411	634	126	144
<i>Discogs</i>	128 070	15	15	270 771	128 070	128 070
<i>CiteSeer</i>	286	385	596	1 653	385	385
<i>Marvel</i>	1 625	111	1 934	9 855	1 625	1 625
<i>DBpedia</i>	2 671	17	2 839	18 517	2 671	2 671
<i>Actor-Movie</i>	294	646	7 799	3 957	646	646
<i>Pics</i>	7 810	335	7 079	113 079	7 810	7 810
<i>YouTube</i>	7 591	1 035	7 357	37 514	7 591	7 591
<i>WikiLens</i>	1 721	80	285	4 826	1 721	1 721
<i>BookCrossing</i>	13 601	2 502	53 916	151 646	13 601	13 601
<i>GitHub</i>	884	3 675	15 995	29 650	3 675	3 675
<i>DailyKos</i>	457	2 123	430	6 895	2 817	2 123
<i>FilmTrust</i>	244	1 044	1 459	1 770	1 100	1 044
<i>CiteULike</i>	4 072	8 814	18 190	80 410	8 814	8 814
<i>Reuters</i>	380	19 044	19 731	37 716	19 044	19 044
<i>BibSonomy</i>	21 463	1 407	4 614	159 465	21 463	21 463
<i>TV-Tropes</i>	6 507	12 400	47 460	37 494	12 400	12 400
<i>DVD-Ciao</i>	34 884	422	13 000	62 027	34 884	34 884

TABLEAU 4.2 – Valeurs maximales des degrés et bidégénérescence des graphes bipartis du tableau 4.1. \mathbf{d}_U et \mathbf{d}_V sont les degrés maximaux dans U et V , \mathbf{d}_{2U} et \mathbf{d}_{2V} sont les degrés projetés maximaux sur U et V , et \mathbf{b}_U et \mathbf{b}_V sont les dégénérescences de U et V .

Graphe	$\overline{d_U}$	$\overline{d_V}$	$\overline{d_{2U}}$	$\overline{d_{2V}}$	$\overline{b_U}$	$\overline{b_V}$
<i>UC-Forum</i>	13	7,9	141	159	91	95
<i>Discogs</i>	32 110	1,8	15	119 244	32 110	102 088
<i>CiteSeer</i>	4,9	2,8	8	57	9,3	43
<i>Marvel</i>	14	7,5	52	1 123	40	769
<i>DBpedia</i>	39	1,5	40	722	47	522
<i>Actor-Movie</i>	11	3,8	321	79	171	47
<i>Pics</i>	58	2,0	259	1 894	173	1 459
<i>YouTube</i>	9,8	3,1	178	1 270	124	1 049
<i>WikiLens</i>	82	5,3	114	1 476	138	1 062
<i>BookCrossing</i>	10	3,4	353	2 340	215	1 669
<i>GitHub</i>	7,8	3,6	789	183	525	103
<i>DailyKos</i>	102	51	3 277	2 273	2 808	1 349
<i>FilmTrust</i>	23	17	1 236	230	1 020	152
<i>CiteULike</i>	23	3,5	4 368	1 332	3 903	915
<i>Reuters</i>	49	25	18 907	503	18 632	287
<i>BibSonomy</i>	78	2,2	750	8 090	584	6 919
<i>TV-Tropes</i>	50	36	10 493	3 531	6 487	2 331
<i>DVD-Ciao</i>	77	22	329	31 398	241	24 195

TABLEAU 4.3 – Valeurs moyennes des degrés et bidégénérescences des graphes bipartis du tableau 4.1. $\overline{d_U}$ et $\overline{d_V}$ sont les degrés moyens dans U et V , $\overline{d_{2U}}$ et $\overline{d_{2V}}$ sont les degrés projetés moyens sur U et V , et $\overline{b_U}$ et $\overline{b_V}$ sont les dégénérescences moyennes de U et V .

degré ou à la bidégénérescence. Dans la plupart des graphes bipartis, notre temps de calcul est meilleur que celui de OOMBEA, et plus le graphe contient de bicliques, plus l'écart s'accentue, gagnant plus d'un ordre de grandeur par exemple pour *FilmTrust* et *CiteULike*. Enfin, pour les graphes contenant le plus de bicliques maximales, l'algorithme OOMBEA ne permet pas d'obtenir l'ensemble des bicliques maximales dans la limite d'une semaine de temps de calcul, même dans les cas de *Reuters* et *BibSonomy* où BBK met moins d'une journée à les énumérer.



Graphes ordonnés par nombre de bicliques maximales croissant

FIGURE 4.4 – Temps de calcul de notre algorithme BBK sur nos jeux de données, et ceux de l'algorithme OOMBEA. À droite, les valeurs pour OOMBEA ne sont pas affichées car le calcul ne s'est pas terminé dans la limite d'une semaine que nous avons fixée.

Néanmoins, il y a tout de même certains graphes bipartis pour lesquels OOMB EA est plus rapide, en particulier parmi les graphes contenant moins de bicliques maximales. C'est le cas de *Discogs*, *DBpedia* et *Actor-Movie*. Il est à noter que pour le graphe *Discogs*, la différence est très élevée. Cela provient de la structure de ce réseau, qui contient peu de sommets dans son ensemble U (seulement 15), mais pour lesquels leur bidégénérescence est très élevée (voir tableau 4.3). Ce graphe est très dense, et il a des degrés projetés extrêmement élevés. Notre algorithme fonctionne bien sur des graphes avec beaucoup de sommets et des degrés faibles en moyenne, mais il semble ne pas être adapté à la situation inverse.

Dans le tableau 4.4, nous avons également affiché le rapport r , noté dans le tableau r_{BBK} , qui apparaît dans la complexité et qui a été défini dans la section 4.2.2. Ce rapport correspond à la fraction des feuilles des arbres des appels récursifs qui renvoient des bicliques maximales. La valeur donnée par $1 - r$ correspond alors à la fraction des feuilles qui correspondent à des branches d'appel, de taille au plus b_U , qui devraient être élaguées par un pivot parfait. On voit que ce rapport est très bon

lors de nos énumérations, ce qui signifie que le pivot que nous avons décrit accomplit bien son rôle pour élaguer les branches sur les graphes bipartis. Le facteur $\frac{1}{r}$ présent dans l'expression de la complexité en fonction de la sortie reste donc faible, ce qui est intéressant en termes de temps de calcul.

Graph	t_{BBK}	t_{OOMBEA}	r_{BBK}
<i>UC-Forum</i>	0.07	0.26	0.60
<i>Discogs</i>	1,185	4.6	1.00
<i>CiteSeer</i>	1.4	2.3	0.71
<i>Marvel</i>	2.1	6.1	0.66
<i>DBpedia</i>	12	7.7	0.40
<i>Actor-Movie</i>	21	19	0.21
<i>Pics</i>	32	34	0.39
<i>YouTube</i>	13	87	0.76
<i>WikiLens</i>	10	111	0.91
<i>BookCrossing</i>	335	6,169	0.77
<i>GitHub</i>	248	7,283	0.84
<i>DailyKos</i>	1,419	35,837	0.77
<i>FilmTrust</i>	20,255	300,307	0.98
<i>CiteULike</i>	12,338	594,549	0.92
<i>Reuters</i>	54,045	-	0.82
<i>BibSonomy</i>	58,719	-	0.96
<i>TV-Tropes</i>	113,659	-	0.73
<i>DVD-Ciao</i>	476,686	-	0.93

TABLEAU 4.4 – Tableau des temps de calculs obtenu par notre algorithme BBK, et par l'algorithme OOMBEA de Chen *et al.* Un symbole “-” signifie que le calcul n'a pas terminé au bout d'une semaine. La dernière colonne représente le rapport r pour l'algorithme BBK défini dans la section 4.2.2 et qui apparaît dans l'expression de la complexité (théorème 4.4).

4.5.3 Énumérer les sommets à partir de U ou de V ?

Dans l'algorithme 4.5, un choix doit être fait pour initialiser les énumérations des bicliques maximales : la boucle de la ligne 1 doit-elle être réalisée sur les sommets de U , ou sur les sommets de V ? Nous montrons ici que ce choix n'est pas anodin et qu'il a un impact significatif sur le temps de calcul que l'on peut anticiper.

Par défaut, nous avons fait le choix de réaliser l'itération sur l'ensemble qui contient le **moins** de sommets. L'ensemble des expériences est réalisée dans ce cas. Nous nous en servons comme expériences de référence. La figure 4.5 montre l'impact du choix de faire l'itération sur l'ensemble contenant le **plus** de sommets, qui est l'ensemble V dans nos jeux de données. Les barres représentées dans cette figure

correspondent au temps relatif calculé à partir de l’itération sur l’ensemble V , divisé par le temps calculé à partir de l’itération sur l’ensemble U . Une barre qui dépasse la ligne $y = 1$ signifie que le temps de calcul est plus lent lorsqu’on réalise l’itération sur l’ensemble de sommets le plus grand, et inversement.

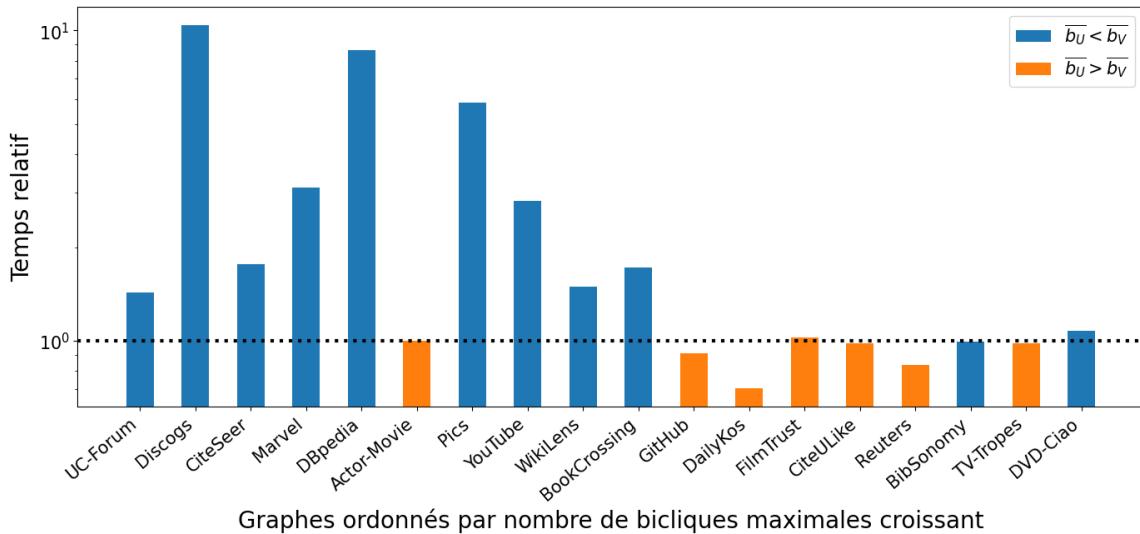


FIGURE 4.5 – Temps d’exécution de l’algorithme 4.5 BBK lorsque l’itération de la ligne 1 se fait sur l’ensemble V où il y a le plus de sommets, divisé par celui où elle se fait sur l’ensemble U où il y en a le moins. Les résultats en bleu correspondent aux graphes pour lesquels $b_U < b_V$, et en orange les graphes où $b_U > b_V$.

On remarque tout d’abord que ce choix peut avoir un fort impact sur le temps de calcul, puisque pour certains graphes comme *Discogs* ou *DBpedia*, le temps de calcul peut être multiplié par 10. Le choix de prendre U le plus petit semble pertinent pour les jeux de données contenant le moins de bicliques maximales, mais ça n’est pas le cas pour les jeux de données en contenant plus. Dans certains cas, il est même meilleur d’itérer sur l’ensemble contenant le plus de sommets. Pour approfondir, nous étudions le temps passé dans la fonction BK-Étendu : nous avons vu que si un sommet u a une bidégénérescence $b(u)$, alors le temps passé dans cette fonction lors de l’itération lui correspondant à la ligne 1 de l’algorithme 4.5 est exponentiel en $b(u)$ (voir théorème 4.3). Pour en comprendre les implications, nous avons coloré les barres, en bleu lorsque c’est $\overline{b_U}$ qui est le plus grand dans le graphe biparti, et en orange lorsque c’est $\overline{b_V}$. Ces valeurs semblent correspondre à ce que l’on observe : il semble intéressant d’itérer les appels à BK-Étendu sur les sommets de l’ensemble U ou V dont la bidégénérescence est la plus faible ; et même s’il y a plus de sommets, si le coût de traitement de ces sommets est plus faible individuellement, cela peut améliorer le temps de calcul général. Il est à noter que, dans notre jeu de données, le fait que $\overline{b_U}$ est le plus grand est équivalent à $\overline{d_{2U}}$ le plus grand. Ainsi, si on veut connaître à l’avance l’ensemble de sommets sur lequel on souhaite faire l’itération

pour exécuter les fonctions `BK-Étendu`, une bonne estimation semble être d'étudier la valeur moyenne du degré projeté de U et de V , qui est plus rapide à calculer que la bidégénérescence. On constate de plus que pour les graphes qui ont le moins de bicliques maximales, \bar{d}_2 et \bar{b} semblent en général plus faibles du côté qui contient le moins de sommets, alors que pour ceux qui contiennent beaucoup de bicliques maximales, ces grandeurs semblent plus faible du côté contenant le plus de sommets. Cela semble traduire le fait qu'une forte densité est plus facile à traiter lorsqu'elle est répartie sur de plus nombreux sommets. Les exceptions que l'on observe sur la figure sont en fait très proches de la valeur 1, ce qui signifie pour ces cas que le choix entre U ou V a peu d'importance.

4.5.4 Impact de l'ordre sur le temps de l'énumération

Un autre facteur qui joue sur le temps d'énumération des bicliques maximales, après avoir choisi l'ensemble de sommets U sur lequel réaliser l'itération de la ligne 1, est l'ordre des sommets dans lequel on réalise cette itération. En effet, pour une itération donnée sur $u \in U$, c'est la taille de l'ensemble P qui détermine le coût principal de l'appel à `BK-Étendu`, puisque cet appel se trouve être exponentiel en $|P|$ [ELS10]. Or, cet ensemble contient les sommets de $N(u) \cup N_2(u) \setminus \{u\}$, auquel on retire ceux qui ont déjà été traités (ligne 2 de l'algorithme 4.5). C'est pour cela que la bidégénérescence est intéressante, car les sommets ayant beaucoup d'éléments dans cet ensemble ont tendance à être traités vers la fin, ce qui fait que l'ensemble P qui leur est associé sera réduit.

La figure 4.6 illustre l'importance de l'ordre dans notre algorithme BBK. On y compare notre algorithme à deux ordres : *l'ordre aléatoire* pour lequel nous donnons le temps de calcul minimal, maximal et médian obtenu sur 20 exécutions pour chacun des graphes (respectivement en bleu, orange et vert), et *l'ordre unicore* utilisé par Chen *et al.* dans leur algorithme (en rouge).

Tout d'abord, nous constatons que les résultats avec l'ordre unicore sont assez proches de ceux avec l'ordre par bidégénérescence car les barres rouges sont proches de la ligne $y = 1$, mais tout de même avec une légère augmentation du temps de calcul (à l'exception des graphes *Pics*, *DailyKos* et *BibSonomy*). Ce n'est pas étonnant, car ces deux ordres sont très proches. En effet, l'ordre unicore calcule un ordre par dégénérescence en considérant $N_2(u) \setminus \{u\}$ comme voisinage augmenté d'un sommet u (*i.e.* le voisinage du graphe projeté), lorsque nous le calculons en considérant $N_2(u) \setminus \{u\} \cup N(u)$. Or, le nombre de voisins projeté étant en général bien plus grand que le nombre de voisins, c'est cette valeur qui domine, d'où le fait que les deux ordres fournissent un résultat similaire.

De plus, l'ordre aléatoire fournit un algorithme globalement plus lent que celui avec l'ordre par bidégénérescence, ce qui justifie la pertinence de l'ordre. Dans tous les cas, le temps médian produit par l'ordre aléatoire est moins bon que l'ordre

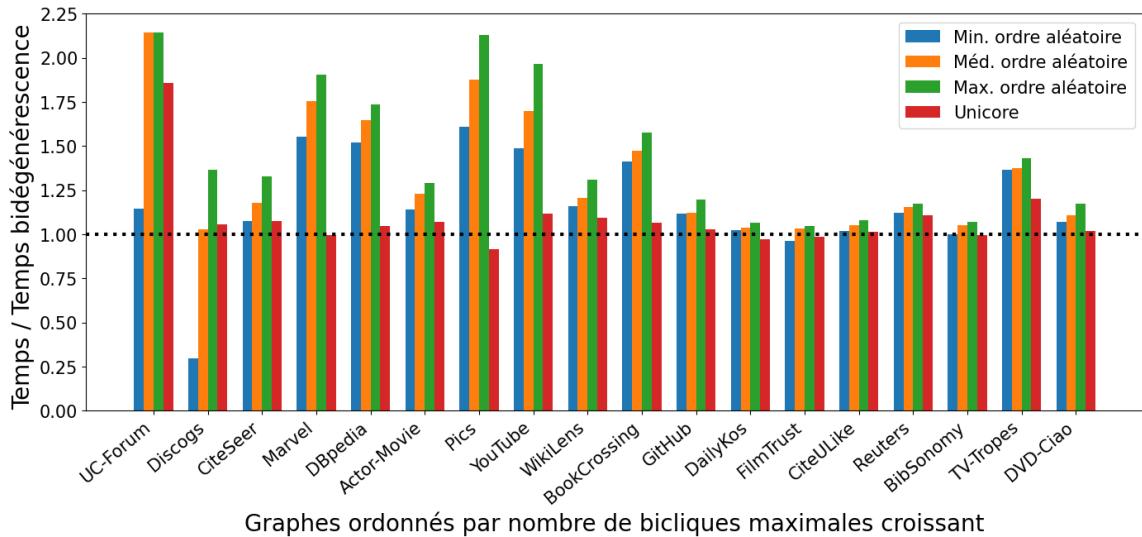


FIGURE 4.6 – Temps relatif de calcul de l’algorithme BBK, pour lequel on remplace l’ordre par bidégénérescence par l’ordre aléatoire et l’ordre unicore utilisé par Chen *et al* [Che+22]. Le temps relatif affiché correspond au temps de l’énumération pour l’ordre correspondant, divisé par le temps pour l’algorithme BBK avec l’ordre par bidégénérescence. L’algorithme avec ordre aléatoire a été exécuté 20 fois, et ses temps de calcul minimal, médian et maximal sont affichés.

par bidégénérescence. De plus, même le temps minimal obtenu est moins bon dans tous les cas, sauf pour les graphes *Discogs* et *FilmTrust*. Pour *Discogs*, le temps minimal observé par l’un des 20 ordres aléatoires exécutés va jusqu’à diviser par quatre le temps de calcul, ce qui illustre que notre algorithme BBK n’est pas adapté pour traiter ce type d’instances très denses. Pour le graphe *FilmTrust*, même si ce n’est une amélioration que de quelques pourcents, cela peut laisser penser qu’il existe une meilleure manière de calculer l’ordre des sommets, par rapport à celle que nous utilisons. Enfin, pour les graphes avec plus de bicliques maximales, l’ordre des sommets semble avoir moins d’influence, puisque l’ordre aléatoire médian est proche de la ligne $y = 1$ dans la plupart des cas.

4.5.5 Discussion sur la mémoire

Bien que notre algorithme BBK soit plus rapide que l’algorithme OOMBEA, l’implémentation d’OOMBEA proposée par Chen *et al.* [Che+22] est plus efficace en mémoire consommée. Pour illustrer cela, la figure 4.7 présente la mémoire vive utilisée par les algorithmes BBK et OOMBEA sur chacun des graphes de nos jeux de données.

La figure 4.7 montre que l’algorithme OOMBEA est capable d’énumérer les bicliques maximales en utilisant environ dix fois moins de mémoire en général, voire

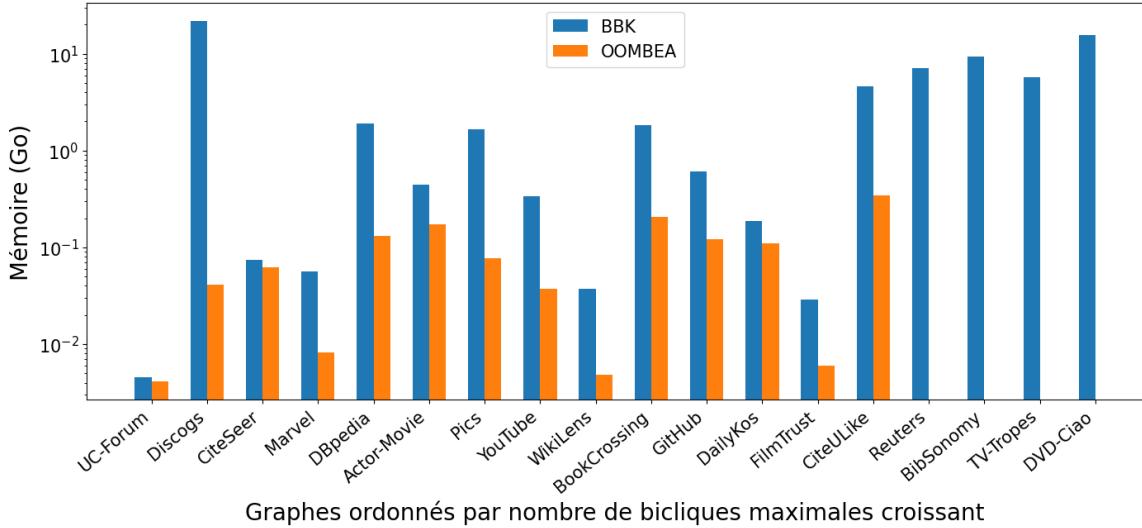


FIGURE 4.7 – Mémoire consommée par les deux algorithmes BBK et OOMBEA. Pour les quatre graphes ayant le plus de bicliques maximales, l’algorithme OOMBEA ne permet pas de terminer le calcul en moins d’une semaine, donc il n’y a pas de résultat affiché.

encore davantage pour le graphe *Discogs*. Cela provient de notre manière d’implémenter l’algorithme : pour gagner en efficacité au sein d’un appel récursif, nous utilisons la méthode décrite par Eppstein *et al.* [ELS10] qui consiste à allouer pour chaque sommet, en plus de l’ensemble de ses voisins, une liste de la taille de sa bidégénérescence. En effet, un sommet ne peut pas appartenir à une biclique contenant plus de sommets que sa bidégénérescence, et c’est donc une borne sur la profondeur de l’arbre des appels à BK-Étendu dans lequel ce sommet peut se trouver. Cette liste sert à pointer l’indice de fin de la liste d’adjacence du sommet, qui diminue à chaque appel récursif. Cette méthode fonctionne bien dans les graphes non bipartis, car la dégénérescence d’un sommet est faible, et même plus faible que le degré du sommet, donc la consommation en mémoire reste linéaire en la taille du graphe. Mais dans les graphes bipartis, comme on l’a vu, la bidégénérescence moyenne des sommets est plus élevée que le degré moyen, donc pour chaque sommet, stocker une liste de la taille de sa dégénérescence entraîne un besoin en mémoire plus élevé.

Néanmoins, sur quasiment tous nos graphes, la mémoire utilisée par l’algorithme BBK ne dépasse pas 2 Go. Ce n’est pas le cas pour nos cinq plus grands graphes, où elle ne dépasse pas 10 Go, à l’exception *DVD-Ciao* qui a besoin de 16 Go. Le pire cas que nous obtenons est celui de *Discogs* qui, comme on l’a vu, est particulier, car ses paramètres de degré et de bidégénérescence sont très élevés. Pour ce graphe, la mémoire nécessaire est de 21 Go. Ainsi, même avec notre algorithme qui est consommateur en mémoire pour les graphes bipartis, nous voyons que la mémoire n’est pas le facteur limitant de l’énumération.

4.6 Conclusion

Dans ce chapitre, nous avons développé un nouvel algorithme pour l'énumération des bicliques maximales dans les graphes bipartis. Pour cela, nous avons identifié une équivalence entre les bicliques d'un graphe biparti et les cliques de son graphe que l'on appelle étendu. Néanmoins, ce graphe étendu est trop dense pour que l'on puisse exécuter l'algorithme classique d'énumération des cliques maximales dans les graphes. Nous avons alors proposé une méthode algorithmique qui permet d'énumérer les cliques maximales de ce graphe, en n'utilisant que le voisinage de son graphe biparti, et pas celui du graphe étendu. Cela nous a permis de profiter du caractère clairsemé des graphes bipartis massifs du monde réel et a abouti à un algorithme très efficace pour l'énumération des bicliques maximales. Dans nos expériences, nous avons montré un gain de temps de calcul sur nos jeux de données massifs, et cela a permis d'obtenir l'ensemble des bicliques maximales sur des graphes pour lesquels l'état de l'art ne termine pas (dans la limite d'une semaine de calcul). Nous avons illustré le fait que l'ordre des sommets ainsi que le choix de l'un des deux ensembles de sommets du graphe biparti sur lequel l'énumération est exécutée ont un impact sur le temps de calcul. En particulier, nous avons constaté que le comportement de l'algorithme n'était pas le même sur les graphes contenant le plus de bicliques maximales et sur ceux en contenant moins, ce qui donne une clé de compréhension sur la structure des graphes bipartis du monde réel. Enfin, nous avons montré dans nos expériences que bien que la mémoire consommée par notre implémentation est plus élevée que dans l'état de l'art, ce n'est pas le facteur limitant pour le calcul, car elle ne dépasse pas 21 Go même sur les plus gros jeux de données testés.

Suite à ces travaux, nous identifions plusieurs perspectives. Tout d'abord, dans l'algorithme BBK, nous avons ordonné les sommets de manière à suivre la logique de l'algorithme de Eppstein *et al.* [ELS10] qui consiste à chercher à minimiser la taille des ensembles P sur lesquels une énumération exponentielle est lancée. Pour cela, nous avons introduit l'ordre par bidégénérescence. Néanmoins, nous avons décrit des améliorations de la fonction BK-Étendu par rapport à l'algorithme de Bron-Kerbosch, concernant le test de maximalité des bicliques. Elles impliquent que le temps de calcul n'est pas nécessairement important pour un ensemble P contenant beaucoup de sommets. En effet, par exemple, si P ne contient qu'un sommet dans U ($|P_U| = 1$), et beaucoup dans V , alors l'algorithme s'exécute en temps linéaire grâce au pivot. En fait, l'algorithme réalise plus de calculs lorsque l'espace de recherche contient beaucoup de sommets à la fois dans U et dans V . Avec cette remarque, il serait intéressant de tester un autre ordre qui traite en priorité les ensembles P dont $\min(|P_U|, |P_V|)$ est faible, plutôt que ceux où $|P_U| + |P_V|$ est faible. Il s'agirait d'un ordre par dégénérescence en considérant la valeur associée à un sommet x égale à $\min(|N(x)|, |N_2(x) \setminus \{x\}|)$. Ceci pourrait permettre d'alléger davantage les ensembles qui contiennent beaucoup de sommets des deux côtés.

En terme d'analyse de complexité, le facteur b^2 apparaissant dans la complexité de sortie (théorème 4.4) n'est pas très satisfaisant. En effet, il s'agit d'une borne sur le coût des opérations au sein du corps d'un appel récursif à **BK-Étendu**. Or, on peut montrer qu'un tel coût est en fait en $\mathcal{O}(|P| \cdot |P \cup X|)$ [ELS10]. Nous pensons que ce facteur est beaucoup moins important que b^2 en pratique, et il serait intéressant de le calculer expérimentalement, afin de raffiner la formule de complexité en fonction de la sortie et voir dans quelle mesure notre algorithme est proche d'une énumération optimale, qui serait du même ordre de grandeur que la taille de la sortie.

Enfin, on pourrait utiliser notre méthode d'énumération des bicliques pour enrichir l'analyse des réseaux massifs issus du monde réel. En particulier, dans le cas des graphes, on a vu que l'énumération des cliques permet de détecter des communautés qui se recouvrent. Cette approche a récemment été utilisée pour la recherche de communautés dans les graphes bipartis [Che+23]. Nous pouvons notamment essayer de nous inspirer des travaux de ce chapitre pour voir dans quelle mesure le graphe étendu que nous avons introduit ici peut directement être utilisé pour la recherche de telles communautés. Le formalisme des hypergraphes est également en plein développement, et il est montré qu'il est équivalent à celui de graphes bipartis. La recherche de cliques dans les hypergraphes a été discutée depuis longtemps, et l'est encore très récemment [ERS77; SZ19; Hol20; LW20; KR23]. Nous pourrions étudier dans quelle mesure il est possible de faire le lien entre les définitions spécifiques aux hypergraphes et ce qu'elles impliquent dans les graphes bipartis, pour voir si des travaux similaires à ceux de ce chapitre peuvent être développés afin d'améliorer l'étude de ces structures complexes.

Énumération des cliques maximales dans les flots de liens réels massifs

Résumé

Dans ce chapitre, nous traitons le problème de l'énumération des cliques maximales dans les flots de liens. Dans un flot de liens, une clique est une paire $(C, [t_0, t_1])$, où C est un ensemble de sommets qui interagissent tous deux-à-deux pendant l'intégralité de l'intervalle de temps $[t_0, t_1]$. Elle est maximale lorsque ni son ensemble de sommets ni son intervalle de temps ne peuvent être augmentés. Certains des principaux travaux existants résolvant ce problème sont basés sur le célèbre algorithme de Bron-Kerbosch pour l'énumération des cliques maximales dans les graphes. Nous partons de cette idée pour proposer un nouvel algorithme d'énumération, qui fait correspondre les cliques du flot de liens et les cliques de ses graphes instantanés (graphes des interactions qui existent à un instant donné). Cela réduit les opérations en ne travaillant que localement à chaque instant, ce qui permet de profiter du caractère peu dense de ces graphes instantanés sur les données du monde réel. Nous prouvons la validité de ce nouvel algorithme, et nous en donnons deux formules de complexité : une en fonction des caractéristiques de l'entrée et une en fonction des caractéristiques de la sortie de l'algorithme. Nous montrons que cette dernière est proche de la taille de la sortie, ce qui montre l'efficacité de l'algorithme. Pour le confirmer, nous réalisons des expériences sur des flots de liens utilisés dans l'état de l'art, et sur des flots de liens massifs issus du monde réel, contenant jusqu'à 100 millions de liens. Dans tous les cas, notre algorithme est plus rapide que les algorithmes existants, et la plupart du temps d'un facteur d'au moins 10 et jusqu'à un facteur de 10^4 . De plus, il permet de traiter flots de liens massifs pour lesquels les autres algorithmes ne sont pas en mesure de fournir une solution.

5.1 Introduction

L'énumération des cliques maximales dans les flots de liens a suscité de l'intérêt ces dernières années, car elle apporte un outil d'analyse puissant au domaine des graphes temporels. Le premier algorithme permettant de réaliser cette tâche a été proposé par Viard *et al.* [VLM16], puis amélioré deux ans plus tard [VML18]. Entre-temps, Himmel *et al.* ont proposé une nouvelle version basée sur l'adaptation de l'algorithme de Bron-Kerbosch à un contexte dynamique [Him+17], qui a ensuite été généralisée à la notion de k -plex temporel par Bentert *et al.* [Ben+19]. Selon les paramètres expérimentaux, l'une ou l'autre méthode peut être plus rapide par rapport aux autres. Cependant, ces méthodes sont toujours limitées à des réseaux dynamiques relativement petits, ne permettant pas l'énumération sur de très grands jeux de données. Il y a donc un besoin d'algorithmes plus efficaces pour lister les cliques maximales dans les flots de liens.

Les contributions de ce chapitre sont les suivantes :

- Nous proposons un nouvel algorithme pour énumérer les cliques maximales dans les flots de liens, capable de traiter des jeux données massifs issus du monde réel.
- Nous analysons la complexité temporelle de cet algorithme : nous en fournissons une qui dépend des caractéristiques de l'entrée, et une qui dépend des caractéristiques de la sortie de l'algorithme. Cette dernière est proche de la taille de la sortie, ce qui montre que l'algorithme est efficace. Nous montrons également que les besoins en mémoire sont proches de l'optimal.
- Nous réalisons des expériences sur des données réelles, qui montrent que l'algorithme est significativement plus rapide que ceux de l'état de l'art. Il permet d'énumérer les cliques maximales dans des réseaux qui sont jusqu'à deux ordres de grandeur plus grands que ce qui était auparavant faisable dans les limites de temps et de mémoire de notre protocole (limitation à 24 heures du temps de calcul et à 390 Go de mémoire vive). Ainsi, nous pouvons énumérer les cliques dans des flots de liens contenant plus de 100 millions de liens, là où il était jusqu'à présent impossible de dépasser un demi-million de liens.
- Enfin, nous fournissons deux implémentations de l'algorithme : l'une en Python, utilisée pour pouvoir se comparer à l'état de l'art qui est codé dans ce langage de programmation, et l'autre en C++, qui est l'implémentation la plus efficace actuellement disponible et propose également une procédure de parallélisation de l'algorithme. Le code est disponible en libre accès¹.

Le reste de ce chapitre est organisé de la manière suivante. La section 5.2 donne les définitions de base et les notations que nous utilisons tout au long du chapitre. La

1. <https://gitlab.lip6.fr/baudin/maxcliques-linkstream>

section 5.3 présente les travaux de la littérature relatifs à l'énumération des cliques maximales dans les flots de liens. Notre nouvel algorithme est ensuite présenté dans la section 5.4. Dans la section 5.5, nous analysons cet algorithme, en donnant une preuve de validité ainsi que les deux complexités théoriques, en fonction de l'entrée et de la sortie. Enfin, dans la section 5.6, nous faisons une évaluation expérimentale des performances de cet algorithme et montrons les résultats de l'implémentation parallèle.

5.2 Définitions et notations

5.2.1 Rappels sur les flots de liens

Nous rappelons ici quelques définitions sur les flots de liens, qui ont été formellement introduites dans le chapitre 2, et qui sont nécessaires pour la suite de ce chapitre. Un ***flot de liens*** est un triplet $L = (T, V, E)$, où T est un intervalle de temps, V un ensemble de sommets et $E \subseteq T \times T \times V \times V$ un ensemble de liens. Un lien $(b, e, u, v) \in E$ signifie que les sommets u et v sont en interaction sur l'ensemble de l'intervalle de temps $[b, e]$. Les liens sont non orientés, *i.e.* (b, e, u, v) et (b, e, v, u) représentent le même lien, et deux liens différents sur des mêmes sommets sont disjoints dans le temps : si $(b, e, u, v) \in E$ et $(b', e', u, v) \in E$ avec $[b, e] \neq [b', e']$, alors $[b, e] \cap [b', e'] = \emptyset$.

Une ***clique*** d'un flot de liens est une paire $(C, [t_0, t_1])$ où $C \subseteq V$ est un ensemble de sommets, avec $|C| \geq 2$, dont chaque paire de sommets est connectée par un lien qui existe durant tout l'intervalle de temps $[t_0, t_1]$. Comme pour une clique dans un graphe, une clique dans un flot de liens peut contenir d'autres cliques. Nous ne nous intéressons ici qu'à celles qui sont maximales. Dans un flot de liens, la notion de maximalité s'applique à la fois en termes de ***temps*** et termes ***sommets***. On dit qu'une clique $(C, [t_0, t_1])$ est ***maximale en temps*** si son intervalle de temps $[t_0, t_1]$ ne peut pas être agrandi, et on dit qu'elle est ***maximale en sommets*** si c'est son ensemble de sommets C qui ne peut pas être agrandi.

Dans ce chapitre, nous nous intéressons aux cliques qui sont maximales à la fois en temps et en sommets : ce sont les cliques ***maximales***. La figure 5.1 (à gauche) représente donne un exemple d'un flot de liens, et de ses quatre cliques maximales : $(\{b, c\}, [1, 5])$, $(\{a, b, c\}, [2, 4])$, $(\{c, d\}, [3, 11])$ et $(\{b, c, d\}, [6, 10])$.

5.2.2 Graphe instantané d'un flot de liens au temps t

Nous donnons à présent quelques définitions qui caractérisent un flot de liens $L = (T, V, E)$ à un instant donné $t \in T$. Les arêtes qui existent à l'instant t peuvent être considérées comme les arêtes d'un graphe que nous appelons le graphe instantané de L au temps t . Plus formellement :

Définition 5.1 (Graphe instantané G_t d'un flot de liens au temps t). *Étant donné un instant $t \in T$, le **graphe instantané G_t** du **flot de liens L** au temps t est le graphe $G_t = (V, E_{G_t})$ tel que :*

$$E_{G_t} = \{\{u, v\} \mid \exists(b, e, u, v) \in E, t \in [b, e]\}.$$

Dans le graphe instantané G_t , chaque arête est induite par un unique lien de L , qui a un temps de fin dans T . Nous formalisons ci-dessous la notion de temps de fin associé à une arête de G_t . Il en découle la définition du temps final d'une clique de G_t , qui correspond au premier instant où l'une de ses arêtes se termine.

Définition 5.2 (Temps de fin $\mathcal{E}_t(u, v)$ d'une arête $\{u, v\}$ de G_t). *Soit $\{u, v\}$ une arête de G_t . Par définition de G_t , il existe un lien unique (b, e, u, v) dans E tel que $t \in [b, e]$. Nous appelons e le **temps de fin de l'arête $\{u, v\}$ dans G_t** et nous le notons $\mathcal{E}_t(u, v)$.*

Définition 5.3 (Temps final $\mathcal{E}_t(C)$ d'une clique C de G_t). *Soit C une clique de G_t . Le **temps final $\mathcal{E}_t(C)$ de la clique C** est le minimum des temps finaux des arêtes de C dans G_t . Formellement :*

$$\mathcal{E}_t(C) = \min_{u, v \in C} \{\mathcal{E}_t(u, v)\}.$$

Par exemple, la figure 5.1 représente un flot de liens (à gauche), et son graphe instantané au temps $t = 3$ (à droite). Ce graphe instantané est le graphe $G_3 = (\{a, b, c, d\}, E_3)$, avec $E_3 = \{\{a, b\}, \{a, c\}, \{b, c\}, \{c, d\}\}$. Les temps de fin de ses arêtes sont $\mathcal{E}_3(a, b) = \mathcal{E}_3(a, c) = 4$, $\mathcal{E}_3(b, c) = 5$ et $\mathcal{E}_3(c, d) = 11$. G_3 contient la clique $\{a, b, c\}$ dont le temps final est 4, correspondant au minimum des temps de fin de ses trois arêtes.

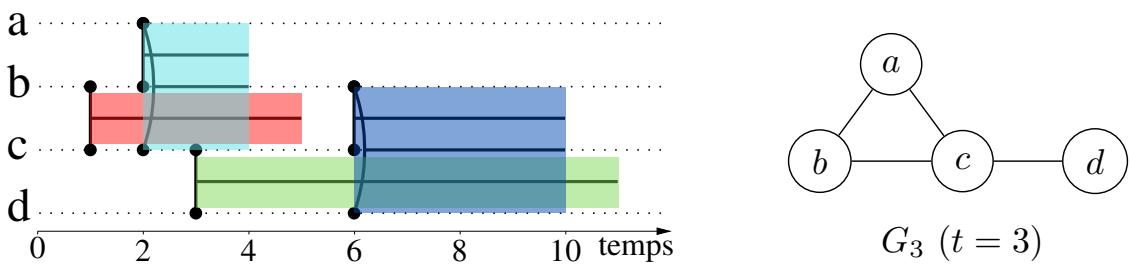


FIGURE 5.1 – **Gauche** : un flot de liens, avec le temps en abscisse et les sommets en ordonnée. Par exemple, il existe un lien entre b et c durant l'intervalle de temps $[1, 5]$. Les cliques maximales sont représentées en couleur. Par exemple, sur l'intervalle de temps $[2, 4]$, les trois sommets a , b et c sont liés entre eux, et forment une clique maximale ($\{a, b, c\}$, $[2, 4]$). **Droite** : le graphe instantané G_3 de ce flot de liens, au temps $t = 3$.

5.3 État de l'art

Dans ce chapitre, nous considérons les travaux qui définissent les cliques dans les flots de liens. Cette question a été abordée avec des formalismes légèrement différents en fonction des articles : certains articles considèrent le formalisme des flots de liens [VLM16 ; VML18] et d'autres utilisent celui des réseaux temporels [Him+17 ; Ben+19]. Tous ces travaux considèrent néanmoins les cliques comme un ensemble de sommets interagissant pendant un intervalle de temps donné, comme nous l'avons défini dans la section 5.2.1. Les deux représentations utilisées dans l'état de l'art sont équivalentes, et il est simple de passer d'une représentation à l'autre, ce qui permet de comparer l'efficacité des différentes méthodes.

Viard *et al.* [VLM16] ont proposé le premier algorithme pour énumérer les cliques maximales dans les flots de liens. Dans ce travail, les liens n'ont pas de durée, mais les cliques sont elles-mêmes paramétrées avec une valeur Δ et sont appelées Δ -cliques. Une Δ -clique $(C, [t_0, t_1])$ est définie de telle manière que chacune des paires de sommets de C interagit au moins une fois pendant chaque sous-intervalle de durée Δ de $[t_0, t_1]$. Leur algorithme a depuis été à la fois simplifié et étendu au formalisme plus général des flots de liens avec durée [VML18]. L'idée sous-jacente à cet algorithme est d'étendre soit en termes de temps, soit en termes de sommets, une clique dans un flot de liens, à partir d'un lien spécifique de départ. Si cette méthode permet effectivement de trouver toutes les cliques maximales du flot de liens, elle nécessite d'enregistrer toutes les cliques en mémoire, afin de décider si une clique donnée a déjà été traitée ou non. Ainsi, cela induit une utilisation importante de la mémoire et limite le calcul dans de nombreux cas, notamment lors du traitement de grands flots de liens.

En parallèle, Himmel *et al.* [Him+16 ; Him+17] ont proposé un autre algorithme pour énumérer les Δ -cliques dans les flots de liens (nommés graphes temporels dans ces articles). Les auteurs adaptent l'algorithme de Bron-Kerbosch à ce cadre dynamique et mettent en œuvre différentes stratégies de sélection du pivot. Cette version surpasse de manière significative l'algorithme de Viard *et al.* de 2016 [VLM16], en particulier pour des valeurs de Δ plus grandes. En revanche, les résultats sont plus contrastés lorsqu'il est comparé à l'algorithme de Viard *et al.* de 2018 [VML18]. Selon le cadre expérimental et les données étudiées, c'est l'une ou l'autre méthode qui peut s'avérer la plus efficace. Plus récemment, une généralisation de cet algorithme a été proposée par Bentert *et al.* [Ben+19] pour énumérer les Δ - k -plexes maximaux dans les flots de liens. Un Δ - k -plex dans un flot de liens est défini par analogie avec un k -plex dans un graphe, qui est un sous-graphe maximal de taille s tel que tout sommet dans le k -plex est adjacent à au moins $s - k$ sommets dans ce sous-graphe. Un Δ - k -plex dans un flot de liens est un sous-ensemble de s noeuds et un intervalle de temps dans lequel chaque noeud interagit avec au moins $s - k$ autres noeuds du Δ - k -plex, au moins une fois pendant chaque sous-intervalle de du-

rée Δ . En particulier, un Δ -1-plex ($k = 1$) est équivalent à une Δ -clique, ce qui permet de comparer cet algorithme aux autres algorithmes décrits dans cette section. C'est un algorithme qui est, lui aussi, basé sur la structure de l'algorithme de Bron-Kerbosch, mais il contient quelques améliorations d'implémentation par rapport à celui de Himmel *et al.* [Him+17], qui le rendent plus efficace en pratique en termes de temps de calcul.

Enfin, certaines méthodes qui ont été conçues à l'origine pour mettre à jour les cliques dans les graphes qui évoluent dans le temps, peuvent être exploitées dans le contexte de l'énumération des cliques dans les flots de liens. Par exemple, Das *et al.* [DST19] appliquent une méthode d'énumération des cliques à ce type de graphes dynamiques. Elle consiste à énumérer, à chaque instant, les cliques qui contiennent au moins une nouvelle arête qui commence à cet instant. L'énumération des cliques contenant des arêtes d'un ensemble donné est une procédure originale qui nécessite des précautions algorithmiques. Bien que le problème formel soit différent du nôtre, cette méthode peut être directement adaptée à notre problème, comme nous le verrons dans la section 5.4.1.

5.4 Un nouvel algorithme d'énumération des cliques maximales dans les flots de liens

Une des idées clés de notre algorithme est de prendre en compte, pour chaque temps donné, uniquement les interactions qui existent à cet instant. Cela présente un intérêt majeur lorsqu'on travaille avec des flots de liens massifs issus du monde réel. En effet, ces flots de liens peuvent avoir une longue période d'existence, mais sont souvent peu denses quand on considère les interactions à un instant donné. Contrairement aux méthodes existantes, qui considèrent le voisinage d'un sommet en englobant toutes les interactions temporelles du flot de liens, notre algorithme se focalise sur des énumérations dans les graphes instantanés. Cette approche permet de réduire la quantité d'opérations à chaque étape, ce qui entraîne des gains significatifs en termes de temps de calcul.

Dans ce qui suit, nous considérons un flot de liens $L = (T, V, E)$ fixé. Nous commençons par décrire la structure générale de notre algorithme (section 5.4.1), puis nous détaillons les deux points principaux : premièrement, nous énumérons les ***cliques maximales en temps*** en énumérant des cliques des graphes instantanés G_t (section 5.4.2), deuxièmement, nous testons leur ***maximalité en sommets*** pour ne garder que celles qui sont maximales (section 5.4.3). Enfin, nous améliorons les temps de calcul en élaguant l'arbre des appels récursifs, avec un ***pivot*** similaire à celui utilisé par Himmel *et al.* [Him+17] (section 5.4.4).

5.4.1 Structure générale de l'algorithme

L'algorithme que nous proposons commence par énumérer exactement une fois chaque clique maximale en temps, puis il filtre parmi elles celles qui sont maximales en sommets, afin d'obtenir l'ensemble des cliques maximales du flot de liens. Pour la première étape, nous utilisons l'équivalence donnée par le théorème 5.1.

Théorème 5.1 (Maximalité en temps d'une clique). *$(C, [t_0, t_1])$ est une clique maximale en temps si et seulement si :*

- (i) *C est une clique du graphe instantané G_{t_0} ;*
- (ii) *Il existe une arête $\{u, v\}$, avec u et v dans C , qui provient d'un lien dont le temps de début est t_0 : $(t_0, e, u, v) \in E$;*
- (iii) *$t_1 = \mathcal{E}_{t_0}(C)$.*

Démonstration. Soit $(C, [t_0, t_1])$ une clique maximale en temps. Alors C est trivialement une clique de G_{t_0} , car au temps t_0 , toutes les paires de sommets de C sont connectées. De plus, Viard *et al.* [VML18] ont prouvé dans leur lemme 3 qu'il existe nécessairement une arête $\{u, v\}$ avec $u, v \in C$ qui provient d'un lien $(t_0, e, u, v) \in E$ commençant à l'instant t_0 . En effet, sinon, tous les liens commencent plus tôt et l'intervalle de temps de la clique peut être étendu à gauche. Enfin, comme, par définition, la clique $(C, [t_0, t_1])$ ne peut pas être étendue dans le temps, cela implique que t_1 doit nécessairement être le temps maximal où ses liens sont tous présents dans le flot de liens, autrement dit $t_1 = \mathcal{E}_{t_0}(C)$.

Réciproquement, si C est une clique du graphe instantané G_{t_0} et $t_1 = \mathcal{E}_{t_0}(C)$, cela signifie que toutes les arêtes de C proviennent de liens du flot qui sont présents durant tout l'intervalle de temps $[t_0, t_1]$. Donc, $(C, [t_0, t_1])$ est une clique de L . De plus, l'intervalle de temps de cette clique est maximal. En effet, par définition de $\mathcal{E}_{t_0}(C)$, au moins un lien n'est plus présent après t_1 . Et, grâce à l'hypothèse selon laquelle il existe une arête $\{u, v\}$ avec $u, v \in C$ qui provient d'un lien $(t_0, e, u, v) \in E$ commençant à t_0 , alors au moins une arête n'est pas présente avant t_0 . Ainsi, la clique $(C, [t_0, t_1])$ est maximale en temps. \square

Nous déduisons du théorème 5.1 que les cliques maximales en temps peuvent être énumérées en parcourant l'ensemble de temps du flot de liens T : pour chaque instant $t \in T$, il suffit d'énumérer les cliques maximales en temps qui commencent au temps t . Pour ce faire, nous énumérons chaque clique (de graphe) C dans G_t , qui contient une arête $\{u, v\}$ d'un lien $(t, e, u, v) \in E$ dont le temps de début est t . Chacune de ces cliques du graphe est alors associée à la clique maximale en temps $(C, [t, \mathcal{E}_t(C)])$ du flot de liens. Alors, le théorème garantit que toutes les cliques maximales en temps sont bien énumérées avec cette méthode. Il ne nous reste plus qu'à trouver celles qui sont également maximales en sommets.

La figure 5.2 illustre cette procédure au temps $t = 3$: les cliques maximales en temps qui commencent à $t = 3$ sont $(\{a, b\}, [3, 7])$, $(\{b, c\}, [3, 5])$ et $(\{a, b, c\}, [3, 5])$, elles contiennent toutes au moins une arête issue d'un nouveau lien, et leur temps de fin correspond au minimum des temps de fin de leurs liens. Seules les cliques $(\{a, b\}, [3, 7])$ et $(\{a, b, c\}, [3, 5])$ sont renvoyées par l'algorithme, car $(\{b, c\}, [3, 5])$ n'est pas maximale en sommets.

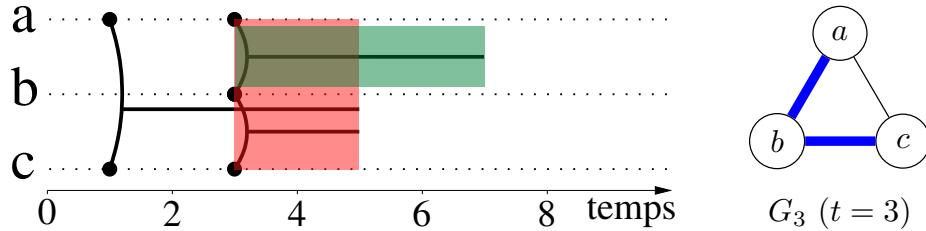


FIGURE 5.2 – **Gauche** : un flot de liens dont les deux cliques maximales qui commencent à $t = 3$ sont affichées en couleur. **Droite** : le graphe instantané de ce flot de liens, G_3 , dont les arêtes qui commencent à $t = 3$ sont colorées bleu.

Le pseudocode est donné dans l'algorithme 5.1, dont la structure générale est résumée dans la Figure 5.3. Nous décrivons tout d'abord les principes de base de l'algorithme 5.1 ; les détails seront abordés dans la suite de cette section. Pour chaque instant $t \in T$ (ligne 1), l'algorithme procède en deux étapes :

- Des lignes 2 à 7, il énumère exactement une fois chaque clique maximale en temps commençant au temps t . Pour ce faire, il énumère les cliques de G_t qui contiennent une arête provenant d'un lien dont le temps de début est t , en traitant chaque lien $(t, _, u, v) \in E$ commençant au temps t (ligne 4). Cette énumération se fait par un appel à la fonction `GraphCliques` (ligne 6). On notera l'utilisation de l'ensemble d'arêtes *ForbidEdges* qui permet d'éviter d'énumérer plusieurs fois une même clique, ce qui sera discuté plus en détail dans la section 5.4.2. En effet, une clique C de G_t peut contenir les sommets de deux liens $(t, _, u_1, v_1)$ et $(t, _, u_2, v_2)$ qui commencent à t , et nous ne voulons pas que cette clique soit énumérée deux fois par le traitement de ces deux liens.
- Dans les lignes 8 à 10, il teste si les cliques maximales en temps énumérées ci-dessus sont maximales en sommets ou non. Pour cela, il utilise le voisinage \mathcal{N}_C de C , c'est-à-dire l'ensemble des voisins communs des sommets de C . Le test de maximalité (ligne 9) sera discuté dans la section 5.4.3.

Notons également que les itérations de la boucle *for* de la ligne 1 de l'algorithme sont indépendantes les unes des autres et peuvent donc être calculées en parallèle. Nous décrivons ce processus de parallélisation et ses résultats dans la section 5.6.

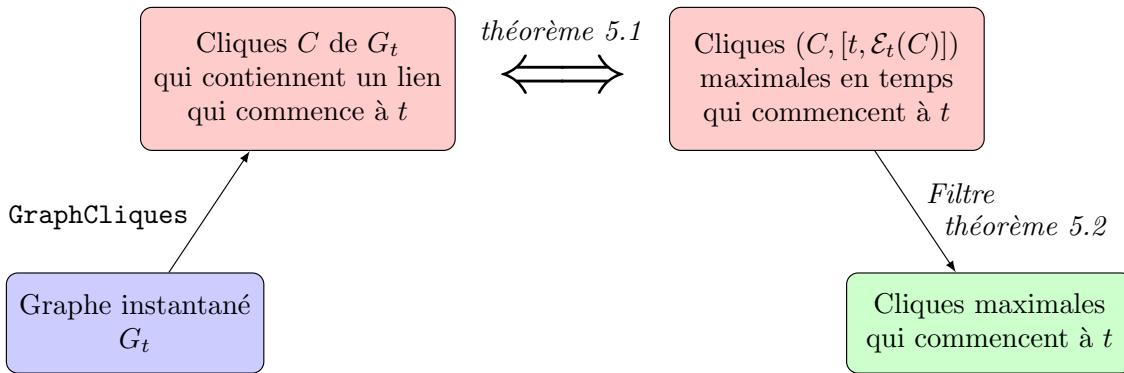


FIGURE 5.3 – Structure générale de l’algorithme 5.1 : pour chaque instant $t \in T$, il se place dans le graphe instantané G_t , où il énumère toutes les cliques qui contiennent un lien qui commence à t . Grâce au théorème 5.1, cela donne l’ensemble des cliques maximales en temps qui commencent à t . Puis, le théorème 5.2 permet d’en extraire l’ensemble des cliques maximales qui commencent à t .

Algorithme 5.1: Énumération des cliques maximales dans les flots de liens.

Entrée: Flot de lien $L = (T, V, E)$.

Sortie: Ensemble des cliques maximales de L .

```

1 for  $t \in T$  do
2    $Cliques \leftarrow \emptyset$ 
3    $ForbidEdges \leftarrow \emptyset$ 
4   for  $(t, \_, u, v) \in E$  do // Liens qui commencent à  $t$ 
5      $P_{uv} \leftarrow N_{G_t}(u) \cap N_{G_t}(v)$ 
6      $Cliques \leftarrow Cliques \cup \text{GraphCliques}(\{u, v\}, P_{uv}, \emptyset, ForbidEdges, t)$ 
7      $ForbidEdges \leftarrow ForbidEdges \cup \{\{u, v\}\}$ 
8   for  $(C, \mathcal{N}_C) \in Cliques$  do //  $\mathcal{N}_C$  = voisinage de la clique  $C$ 
9     if  $\forall u \in \mathcal{N}_C, \mathcal{E}_t(C \cup \{u\}) < \mathcal{E}_t(C)$  then // Test de maximalité
10      output  $(C, [t, \mathcal{E}_t(C)])$ 

11 Fonction GraphCliques( $R, P, X, ForbidEdges, t$ ):
12   output  $(R, P \cup X)$  //  $P \cup X$  = voisinage de la clique  $R$ 
13    $Q \leftarrow \{u \in P \mid \exists v \in R, \{u, v\} \in ForbidEdges\}$ 
14   for  $u \in P \setminus Q$  do
15     GraphCliques( $R \cup \{u\}, P \cap N_{G_t}(u), X \cap N_{G_t}(u), ForbidEdges, t$ )
16      $P \leftarrow P \setminus \{u\}$ 
17      $X \leftarrow X \cup \{u\}$ 
  
```

5.4.2 Énumération des cliques dans les graphes instantanés G_t

L'algorithme 5.1 doit énumérer, pour chaque instant $t \in T$, l'ensemble des cliques du graphe instantané G_t qui contiennent un lien commençant à t . Cette énumération est réalisée par la boucle de la ligne 4, grâce aux appels à `GraphCliques` de la ligne 6. Chacun de ces appels énumère l'ensemble des cliques de G_t qui contiennent l'arête $\{u, v\}$ associée au lien $(t, _, u, v)$ de l'itération courante, et qui ne contiennent aucune arête des itérations précédentes. Pour cela, nous utilisons l'algorithme de Bron-Kerbosch, adapté de telle manière qu'il n'énumère aucune clique contenant une arête des itérations précédentes. Cette procédure particulière nécessite une précaution algorithmique, qu'on peut voir mise en place par Das *et al.* [DST19]. Nous l'adaptons également pour que chaque clique en construction R soit renvoyée, qu'elle soit maximale ou non dans G_t .

Plus précisément, chaque appel à `GraphCliques`($R, P, X, ForbidEdges, t$) de l'algorithme 5.1 énumère l'ensemble des cliques C du graphe instantané G_t telles que $R \subseteq C \subseteq R \cup P$ et qui ne contiennent aucune arête de $ForbidEdges$. Comme pour l'algorithme de Bron-Kerbosch, ici, la clique en construction est R , son voisinage est $P \cup X$ (c'est-à-dire les voisins communs à tous les sommets de R), et parmi eux, les candidats pour faire grossir R sans énumérer de doublon sont les sommets de P . Le fait que $P \cup X$ est le voisinage de la clique R est formellement démontré dans la Section 5.5.1 par le lemme 5.1. Il convient de noter trois différences majeures avec l'algorithme de Bron-Kerbosch, qui a été détaillé dans l'algorithme 4.1 du chapitre 4, à la section 4.2 :

- chaque clique R est renvoyée par la fonction avec son voisinage $P \cup X$ (ligne 12), car il est nécessaire au test de maximalité en sommets de la ligne 9, comme discuté dans la section 5.4.3 ;
- une clique est renvoyée par la fonction, qu'elle soit maximale dans G_t ou non (ligne 12) : il n'y a pas de test équivalent au test de maximalité de Bron-Kerbosch (où la clique R n'est renvoyée que si $P \cup X = \emptyset$) ;
- les cliques de sortie ne doivent contenir aucune arête de $ForbidEdges$; ceci est assuré par les lignes 13 et 14, qui empêchent d'ajouter un sommet u à R si cela implique l'ajout d'une arête $\{u, v\}$ de $ForbidEdges$ à la clique résultante $R \cup \{u\}$.

L'utilisation de l'ensemble $ForbidEdges$ permet de garantir que chaque clique est énumérée au plus une fois, et on le démontre formellement dans la section 5.5. Cet ensemble est nécessaire, car si une clique C contient deux arêtes $\{u_1, v_1\}$ et $\{u_2, v_2\}$, alors elle appartient à l'ensemble des cliques de G_t qui contiennent $\{u_1, v_1\}$, mais aussi à l'ensemble des cliques qui contiennent $\{u_2, v_2\}$. Ainsi, si `GraphCliques` est appelée à la ligne 6 au même instant t sur $\{u_1, v_1\}$ puis $\{u_2, v_2\}$, mais sans la

précaution d'interdire les arêtes de *ForbidEdges*, alors C serait énumérée une fois par appel, et donc deux fois. Ce problème est illustré par l'exemple de la Figure 5.4 qui représente un graphe instantané G_t , dont les arêtes en rouge correspondent à deux nouveaux liens au temps t , et donc à deux appels à **GraphCliques**. La clique $\{a, b, c, d\}$ contient chacune de ces deux arêtes et l'ensemble *ForbidEdges* est utilisé pour qu'elle ne soit pas énumérée deux fois. Ainsi, toutes les cliques contenant au moins une nouvelle arête (arête rouge) sont énumérées exactement une fois.

5.4.3 Filtrer les cliques maximales en sommets parmi les cliques maximales en temps

Une fois que toutes les cliques maximales en temps de L ont été énumérées, grâce au théorème 5.1 et à la fonction **GraphCliques**, elles ne doivent pas toutes être renvoyées en sortie de l'algorithme 5.1. En effet, certaines ne sont pas maximales, car elles ne sont pas maximales en sommet. Par exemple, dans la Figure 5.2, la clique maximale en temps $(\{b, c\}, [3, 5])$ n'est pas maximale en sommets, car on peut y ajouter le sommet a sans réduire son temps final. En revanche, ce n'est pas le cas pour la clique $(\{a, b\}, [3, 7])$, ni pour la clique $(\{a, b, c\}, [3, 5])$, qui sont toutes les deux à la fois maximale en temps et en sommets, et donc maximales.

Ainsi, nous mettons en place un test de maximalité, permettant de tester si une clique maximale en temps est maximale en sommets ou non. Ce test correspond au théorème 5.2, qui donne une condition nécessaire et suffisante pour qu'une clique maximale en temps soit maximale en sommets. Il est appliqué par la ligne 9 de l'algorithme, qui filtre ainsi l'ensemble des cliques maximales, parmi l'ensemble des cliques maximales en temps.

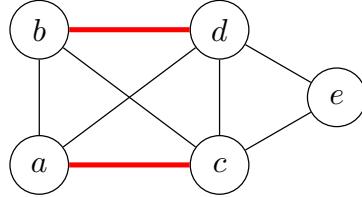
Théorème 5.2 (Maximalité en sommets d'une clique maximale en temps). *Soit $(C, [t, \mathcal{E}_t])$ une clique maximale en temps et \mathcal{N}_C le voisinage de la clique C dans G_t (i.e. $\mathcal{N}_C = \bigcap_{v \in C} N_{G_t}(v)$). Alors :*

$$(C, [t, \mathcal{E}_t(C)]) \text{ est maximale en sommets} \Leftrightarrow \forall u \in \mathcal{N}_C, \quad \mathcal{E}_t(C \cup \{u\}) < \mathcal{E}_t(C).$$

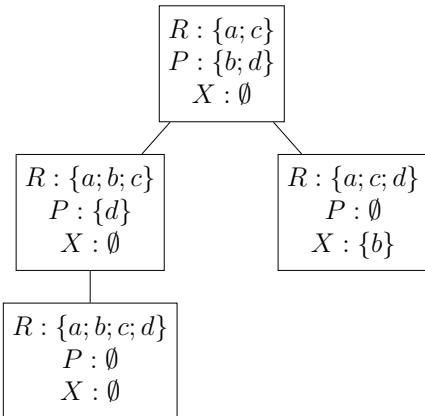
En d'autres termes, une clique maximale en temps $(C, [t, \mathcal{E}_t(C)])$ est maximale en sommets (et donc maximale) lorsqu'on ne peut ajouter aucun sommet à C sans réduire son temps final. Notez que si le voisinage de C est vide, c'est-à-dire $\mathcal{N}_C = \emptyset$, alors $(C, [t, \mathcal{E}_t(C)])$ est toujours maximale en sommets.

Démonstration. Démontrons l'équivalence des négations.

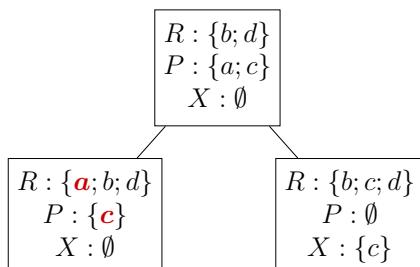
Premièrement, soit $(C, [t, \mathcal{E}_t(C)])$ une clique, et supposons qu'il existe un certain $u \in \bigcap_{v \in C} N_{G_t}(v)$ tel que $\mathcal{E}_t(C \cup \{u\}) \geq \mathcal{E}_t(C)$. Alors, par définition de $\mathcal{E}_t(C \cup \{u\})$, pour chaque paire de sommets $x, y \in C \cup \{u\}$, $x \neq y$, il existe un lien $(b, e, x, y) \in E$

 $G_t, t \in T$

- (a) Exemple d'un graphe instantané G_t d'un flot de liens L à un instant t . Les arêtes colorées en rouge correspondent à des liens qui commencent au temps t , tandis que les autres arêtes correspondent à des liens qui ont commencé avant t . La clique $\{a, b, c, d\}$ contient les deux nouvelles arêtes $\{a, c\}$ et $\{b, d\}$, mais elle ne doit être énumérée qu'une seule fois.



- (b) Arbre des appels à `GraphCliques` réalisé sur la première arête, $\{a, c\}$, et pour lequel $\text{ForbidEdges} = \emptyset$.



- (c) Arbre des appels à `GraphCliques` réalisé sur la deuxième arête, $\{b, c\}$, et pour lequel $\text{ForbidEdges} = \{\{a, c\}\}$. Comme $a \in R$, $c \in P$ et $\{a, c\} \in \text{ForbidEdges}$, alors $c \in Q$. L'appel sur c en bas à gauche mènerait à un deuxième appel avec $R = \{a, b, c, d\}$ et il n'est pas réalisé, grâce à ForbidEdges .

FIGURE 5.4 – Exemple d'une énumération de cliques où l'ensemble ForbidEdges est nécessaire. Chaque noeud des arbres des appels correspond à une clique qui est renvoyée. On voit dans ces arbres que chaque clique contenant l'une des deux nouvelles arêtes du graphe de la figure (a) est énumérée exactement une fois.

tel que $[t, \mathcal{E}_t(C)] \subseteq [b, e]$. Par conséquent, $(C \cup \{u\}, [t, \mathcal{E}_t(C)])$ est une clique de L , et donc $(C, [t, \mathcal{E}_t(C)])$ n'est pas maximale (car $u \notin C$, et donc $C \subsetneq C \cup \{u\}$).

Réciproquement, supposons que $(C, [t, \mathcal{E}_t(C)])$ n'est pas maximale en sommets. Alors, par définition, il existe un sommet $u \in \bigcap_{v \in C} N_{G_t}(v)$ tel que $(C \cup \{u\}, [t, \mathcal{E}_t(C)])$ est une clique de L . Alors, $\mathcal{E}_t(C \cup \{u\}) \geq \mathcal{E}_t(C)$. \square

Par conséquent, le test effectué à la ligne 9 correspond bien au test de maximalité du théorème 5.2 et il permet à l'algorithme 5.1 de bien renvoyer uniquement les cliques qui sont maximales.

5.4.4 Pivot pour améliorer l'énumération des cliques dans les graphes G_t

L'algorithme de Bron-Kerbosch dans les graphes est généralement implémenté avec un pivot qui permet d'élaguer l'arbre des appels récursifs, en supprimant une partie des appels sans modifier la sortie. Un processus similaire peut être mis en œuvre dans le contexte dynamique pour éliminer certains des appels récursifs lors de l'énumération des cliques maximales en temps. Cela a été proposé par Himmel *et al.* [Him+17]. Nous adaptons cette amélioration dans la fonction `GraphCliques` de l'algorithme 5.1 en introduisant la fonction `GraphCliquesPivot` dans l'algorithme 5.2. Nous avons mis en couleur les modifications ajoutées par rapport à la fonction `GraphCliques`.

Algorithme 5.2: Pivot pour énumérer les cliques maximales de L .

Entrée: Flot de lien $L = (T, V, E)$.

Sortie: Ensemble des cliques maximales de L .

1 *Algorithme 5.1 en remplaçant la fonction GraphCliques par :*

2 Fonction `GraphCliquesPivot`($R, P, X, ForbidEdges, t$):
3 **output** ($R, P \cup X$) // $P \cup X$ = voisinage de la clique R
4 $Q \leftarrow \{u \in P \mid \exists v \in R, \{u, v\} \in ForbidEdges\}$
5 **p** \leftarrow pivot $\in P \cup X$
6 **Del** $\leftarrow \{u \in P \cap N_{G_t}(p) \mid \mathcal{E}_t(R \cup \{u, p\}) = \mathcal{E}_t(R \cup \{u\})\}$
7 **for** $u \in (P \setminus Del) \setminus Q$ **do**
8 `GraphCliquesPivot`($R \cup \{u\}, P \cap N_{G_t}(u), X \cap N_{G_t}(u)$,
9 $ForbidEdges, t$)
10 $P \leftarrow P \setminus \{u\}$
11 $X \leftarrow X \cup \{u\}$

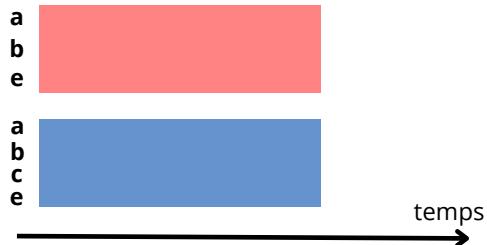
Selon la procédure de l'algorithme 5.2, les sommets de l'ensemble *Del* sont les sommets sur lesquels on ne fait pas d'appel récursif. La figure 5.5 illustre cette

procédure, en montrant un exemple d'un sommet de $P \cap N_{G_t}(p)$ qui appartient à Del , et un exemple qui n'appartient pas à Del .

$$R = \begin{array}{c} (a) \\ \text{---} \\ (b) \end{array}$$

$$P = \begin{array}{c} \text{pivot} \\ (c) \\ \diagdown \quad \diagup \\ (d) \quad (e) \end{array}$$

- (a) État d'un appel à **GraphCliques** avec $R = \{a, b\}$ et $P = \{c, d, e\}$ et $X = \emptyset$, dont les sous-graphes induits de R et P sont représentés. Le choix du pivot est c , et l'ensemble Del contient au plus les sommets d et e , sur lesquels il n'y aurait pas besoin de faire d'appel récursif.



- (b) Durée de la clique $\{a, b, e\}$ par rapport à la clique $\{a, b, c, e\}$. Elles ont le même intervalle de temps d'existence, donc la clique sur $\{a, b, e\}$ n'est pas maximale en sommets : il n'y a pas besoin de faire un appel récursif sur $e : e \in Del$.



- (c) Durée de la clique $\{a, b, d\}$ par rapport à la clique $\{a, b, c, d\}$. La clique $\{a, b, d\}$ dure strictement plus longtemps, donc il y a besoin de faire un appel récursif sur $d : d \notin Del$.

FIGURE 5.5 – Exemple d'une procédure d'élagage des appels récursifs par le pivot donné dans l'algorithme 5.2. La figure (a) représente l'état d'un appel récursif, avec un pivot c et deux sommets pour lesquels il faut tester s'ils sont dans Del ou non. La figure (b) présente le cas où $e \in Del$, et la figure (c) le cas où $d \notin Del$.

Ne pas faire d'appel récursif pour ces sommets ne supprime aucune clique maximale de la sortie de l'énumération. La preuve formelle est donnée dans le théo-

rème 5.4, dans la section 5.5.1. Sans le démontrer formellement ici, nous en donnons une intuition : à un instant donné t , pour toute clique maximale $(C, [t, \mathcal{E}_t(C)])$ telle que $R \subsetneq C \subseteq R \cup P$, et étant donné un pivot $p \in P \cup X$, il existe toujours un sommet $u \in P \setminus Del$ qui permet d'étendre R vers C . Pour observer cela, distinguons deux cas :

- s'il existe un sommet $u \in C$ qui n'est pas voisin de p , alors $u \notin Del$ puisque $Del \subseteq N_{G_t}(p)$, et $u \in P$ puisque $R \subseteq N_{G_t}(p)$. Alors, un appel récursif avec u étend R vers C (notons que si $p \in C$ alors $u = p$ puisque $p \notin N_{G_t}(p)$),
- sinon, tous les sommets de C sont voisins de p , ce qui signifie d'une part que $p \notin C$, et d'autre part que $C \cup \{p\}$ est une clique de G_t . Puisque $(C, [t, \mathcal{E}_t(C)])$ est maximale, cela implique que $\mathcal{E}_t(C \cup \{p\}) < \mathcal{E}_t(C)$. Il doit donc exister un sommet $u \in P \cap C$ tel que $\mathcal{E}_t(R \cup \{u, p\}) < \mathcal{E}_t(R \cup \{u\})$, i.e. u n'appartient pas à Del et permet d'étendre R vers C .

En outre, pour l'énumération des cliques dans les graphes, il est établi que la meilleure stratégie de pivot est celle qui maximise l'ensemble de sommets à éliminer des appels récursifs [TTT06]. De la même manière, nous choisissons un pivot qui permet de couper autant d'appels que possible. Au sein d'un appel à `GraphCliquesPivot`, pour chaque potentiel pivot $p \in P \cup X$, nous choisissons celui qui maximise la taille de l'ensemble Del . Il a été montré expérimentalement par Himmel *et al.* [Him+17] que ce choix permet effectivement d'obtenir l'implémentation la plus efficace, parmi les différentes stratégies testées. Nous évaluons dans la section 5.6.4 l'efficacité de cet élagage, en comparant les calculs effectués avec et sans pivot.

5.5 Analyse de l'algorithme

Pour comprendre plus en détail le fonctionnement et le comportement de nos algorithmes sans pivot (5.1) et avec pivot (5.2), nous en proposons ici une analyse formelle détaillée. Nous commençons par présenter leur validité dans la section 5.5.1, en démontrant d'abord que l'algorithme 5.1 énumère bien l'ensemble des cliques maximales du flot de liens d'entrée (théorème 5.3), puis que l'ajout du pivot par l'algorithme 5.2 ne modifie pas sa sortie (théorème 5.4). Ensuite, dans la section 5.5.2, nous proposons une analyse de leur complexité temporelle (théorème 5.5) et mémoire (théorème 5.6), et nous donnons en plus une expression de la complexité temporelle en fonction des caractéristiques de la sortie de l'algorithme (théorème 5.7). Cette dernière introduit un facteur $\frac{1}{r}$ qui nous permettra de mesurer expérimentalement l'efficacité du pivot introduit par l'algorithme 5.2, par rapport à l'algorithme 5.1.

5.5.1 Validité

Validité de l'énumération des cliques maximales sans pivot

Pour montrer la validité de l'algorithme sans pivot (5.1), nous avons besoin des lemmes préliminaires suivants. Le lemme 5.1 donne des propriétés sur les éléments R et $P \cup X$ sortis par les appels à `GraphCliques` dans l'algorithme, et le lemme 5.2 montre que chaque clique maximale qui commence au temps t est bien énumérée (exactement une fois) par la procédure des lignes 2 à 7. Enfin, la validité de l'algorithme est démontrée par le théorème 5.3.

Lemme 5.1. *Dans l'algorithme 5.1, à chaque fois qu'un appel à `GraphCliques`($R, P, X, ForbidEdges, t$) est réalisé, ses arguments vérifient :*

- R est une clique du graphe instantané G_t ;
- $P \cup X = \bigcap_{v \in R} N_{G_t}(v)$, i.e. $P \cup X$ contient l'ensemble des voisins communs des sommets de R .

Démonstration. `GraphCliques` étant une fonction récursive, nous démontrons ce lemme par induction sur les arbres des appels récursifs effectués par l'algorithme 5.1, qui sont initialisés aux itérations de la ligne 6.

Initialisation. Le premier appel d'une série d'appels récursifs, effectué à la ligne 6 de l'algorithme 5.1, est de la forme `GraphCliques`($\{u, v\}, N_{G_t}(u) \cap N_{G_t}(v), \emptyset, ForbidEdges, t$), pour deux sommets donnés $u, v \in V$. À ce stade, on a $R = \{u, v\}$, et u et v sont connectés dans G_t , car ils proviennent d'un lien $(t, _, u, v)$ de E (ligne 4). Donc R est bien une clique de G_t . De plus, $P = N_{G_t}(u) \cap N_{G_t}(v)$ et $X = \emptyset$, donc $P \cup X = \bigcap_{v \in R} N_{G_t}(v)$.

Induction. Supposons que `GraphCliques`($R, P, X, ForbidEdges, t$) vérifie que R est une clique de G_t et $P \cup X = \bigcap_{v \in R} N_{G_t}(v)$. Montrons alors que les appels récursifs faits à la ligne 15 de cet appel conservent ces propriétés sur leurs arguments induits. D'une part, il est à noter que $P \cup X = \bigcap_{v \in R} N_{G_t}(v)$ est un invariant de la boucle de la ligne 14 : en effet, après les opérations des lignes 16 et 17, $P \leftarrow P \setminus \{u\}$ et $X \leftarrow X \cup \{u\}$, donc $P \cup X$ n'est pas modifié. Considérons alors une itération de cette boucle donnée, associée au sommet u . L'appel récursif est effectué sur $R' = R \cup \{u\}$, $P' = P \cap N_{G_t}(u)$ et $X' = X \cap N_{G_t}(u)$. Premièrement, $u \in P$, donc par hypothèse d'induction, u est voisin de tous les sommets de R , donc R' est une clique de G_t . Deuxièmement, $\bigcap_{v \in R'} N_{G_t}(v) = \bigcap_{v \in R} N_{G_t}(v) \cap N_{G_t}(u) = (P \cup X) \cap N_{G_t}(u) = (P \cap N_{G_t}(u)) \cup (X \cap N_{G_t}(u))$, donc $\bigcap_{v \in R'} N_{G_t}(v) = P' \cup X'$. Ainsi, les propriétés sont effectivement vraies à chaque appel récursif.

□

Lemme 5.2. Pour $t \in T$, l'ensemble Cliques parcouru à ligne 8 de l'algorithme 5.1 contient exactement une paire (C, \mathcal{N}_C) par clique maximale en temps $(C, [t, \mathcal{E}_t(C)])$ de L qui commence à t .

Démonstration. Soit $(C, [t, \mathcal{E}_t(C)])$ une clique maximale en temps de L , qui commence à l'instant t (notez qu'une clique maximale en temps qui commence à t est toujours de cette forme d'après le théorème 5.1). On montre par induction sur $2 \leq k \leq |C|$ la propriété suivante, $\mathcal{P}(k)$: “Dans l'algorithme 5.1, il existe un appel à $\text{GraphCliques}(R, P, X, \text{ForbidEdges}, t)$ pour lequel R est tel que $|R| = k$, $R \subseteq C \subseteq R \cup P$ et ForbidEdges ne contient aucune arête de C .”

Initialisation. D'après le théorème 5.1 (ii), il existe au moins un lien (t, e, u, v) dans le flot de liens qui commence à l'instant t avec $u, v \in C$. Considérons le premier qui est traité par la boucle de la ligne 4. Considérons ensuite l'appel qui est fait à GraphCliques dans cette itération, à la ligne 6. Dans cet appel, du fait d'avoir choisi le premier tel lien, ForbidEdges ne contient aucune arête de C . De plus, $R = \{u, v\}$ et $P = N_{G_t}(u) \cap N_{G_t}(v)$. Or, comme C est une clique, tous ses éléments différents de u et v , sont voisins de u et v . On a donc $R \subseteq C \subseteq R \cup P$. Ainsi, $\mathcal{P}(2)$ est vraie.

Induction. Soit k tel que $2 \leq k < |C|$ (on peut supposer $|C| \geq 3$, car le cas où $|C| = 2$ est traité par $\mathcal{P}(2)$). Supposons que $\mathcal{P}(k)$ est vraie, et montrons $\mathcal{P}(k+1)$. Soit $\text{GraphCliques}(R, P, X, \text{ForbidEdges}, t)$ un appel de l'algorithme 5.1 correspondant à $\mathcal{P}(k)$. D'après $\mathcal{P}(k)$, nous savons que ForbidEdges ne contient aucune arête de C , donc Q , défini à la ligne 13, vérifie $C \cap Q = \emptyset$. Donc, les sommets de $C \setminus R$, qui sont les sommets pouvant faire grossir R , se trouvent dans $P \setminus Q$. Sans perte de généralité, considérons le premier sommet $u \in C \setminus R$ traité par la boucle de la ligne 14 (il y en a bien un car $k < |C|$), et l'appel récursif correspondant cette itération, $\text{GraphCliques}(R \cup \{u\}, P \cap N_{G_t}(u), X \cap N_{G_t}(u), \text{ForbidEdges}, t)$. Dans cet appel, ForbidEdges est inchangé, donc il ne contient aucune arête de C . De plus, comme C est une clique, les sommets de C différents de u sont tous inclus dans $N_{G_t}(u)$, donc $C \subseteq R \cup P$ peut se réécrire $C \subseteq (R \cup \{u\}) \cup (P \cap N_{G_t}(u))$. En outre, $R \cup \{u\} \subseteq C$, et $|R \cup \{u\}| = k + 1$. Par conséquent, $\mathcal{P}(k+1)$ est vraie.

Ainsi, en appliquant $\mathcal{P}(k)$ au cas où $k = |C|$, on obtient qu'il existe un appel dans l'algorithme 5.1 qui renvoie (C, \mathcal{N}_C) au temps t (à la ligne 12). Montrons maintenant qu'il ne peut pas exister d'autre appel renvoyant (C, \mathcal{N}_C) au temps t . Pour cela, considérons la première itération de la ligne 4 impliquant un lien (t, e, u, v) tel que $u, v \in C$. Alors, après cette itération, l'arête $\{u, v\}$ est ajoutée à ForbidEdges (ligne 7). Or, comme Q (ligne 13) empêche la clique en construction de contenir

une arête de *ForbidEdges*, aucun appel ne peut énumérer C dans les itérations suivantes. De plus, comme il s'agit du premier appel impliquant un lien contenant une arête de C , (C, \mathcal{N}_C) n'a pas pu être énumérée lors des itérations précédentes. Enfin, si nous considérons cette itération, l'appel à **GraphCliques** ne peut pas énumérer (C, \mathcal{N}_C) deux fois, car chaque appel récursif dans la boucle commençant à la ligne 14 énumère des cliques différentes, puisque le sommet u utilisé pour faire grossir R est directement supprimé de P à la ligne 16. \square

Théorème 5.3 (Validité de l'algorithme sans pivot). *L'algorithme sans pivot (5.1) énumère une et une seule fois chaque clique maximale du flot de liens d'entrée L .*

Démonstration. Montrons que pour tout $t \in T$, l'itération de la boucle de la ligne 1 qui correspond à t énumère une fois chaque clique maximale qui commence à t , et aucune autre. Selon le lemme 5.2, la boucle qui commence à la ligne 8 itère sur tous les couples (C, \mathcal{N}_C) tels que $(C, [t, \mathcal{E}_t(C)])$ est une clique maximale en temps qui commence à t , et sans répétition. De plus, d'après le lemme 5.1, $\mathcal{N}_C = P \cup X = \bigcap_{v \in C} N_{G_t}(v)$ donc, d'après le test de maximalité en sommets du théorème 5.2, les cliques maximales en temps qui sont renvoyées après le test de la ligne 9 sont toutes celles qui sont maximales en sommets, et donc maximales. Ainsi, à l'itération t , l'algorithme renvoie une fois toutes les cliques maximales qui commencent à t , et aucune autre. Comme la boucle de la ligne 1 couvre tous les instants t où un lien commence, elle couvre tous les instants où une clique maximale peut commencer (théorème 5.1) : on en déduit que toutes les cliques maximales sont bien énumérées par l'algorithme 5.1. \square

Validité de l'énumération avec pivot

Dans la section 5.4.4, nous avons proposé d'introduire un pivot qui permet d'élaguer l'arbre des appels récursifs de **GraphCliques**, en définissant la fonction **GraphCliquesPivot** (algorithme 5.2). En coupant les branches correspondant à des calculs redondants, cet algorithme améliore les temps d'exécution pratiques sans changer la sortie de l'énumération. Nous démontrons ici que cet algorithme est correct.

Théorème 5.4 (Validité de l'algorithme avec pivot). *L'algorithme avec pivot (5.2) est correct : la sortie est la même que celle de l'algorithme sans pivot (5.1).*

Démonstration. Le pivot de la fonction **GraphCliquesPivot** ne peut que réduire l'ensemble de nœuds candidats à faire grossir la clique R en cours de construction, par rapport à la fonction **GraphCliques**. Il ne peut donc que faire réduire l'ensemble *Cliques* de la ligne 8 de l'algorithme 5.1. On peut donc affirmer que toutes les cliques renvoyées par l'algorithme avec pivot sont toujours des cliques maximales de L et qu'elles sont toutes distinctes.

Il reste à montrer que chaque clique maximale est effectivement renvoyée par l'algorithme avec pivot (5.2). Soit $(C, [t, \mathcal{E}_t(C)])$ une clique maximale. En suivant le même raisonnement que celui que nous avons utilisé pour la preuve du lemme 5.2, nous montrons par récurrence sur $2 \leq k \leq |C|$ la propriété $\mathcal{P}_p(k)$: “Il existe un appel `GraphCliquesPivot(R,P,X,ForbidEdges,t)`, tel que R satisfait $|R| = k$ avec $R \subseteq C \subseteq R \cup P$ et $ForbidEdges$ ne contient aucune arête de C ”.

L'étape d'initialisation est la même que dans la preuve du lemme 5.2. Maintenant, en supposant que $\mathcal{P}_p(k)$ est vraie, étant donné $k < |C|$ et `GraphCliquesPivot(R,P,X,ForbidEdges,t)` l'appel correspondant, il suffit de montrer qu'il existe un sommet de $C \setminus R$ pour étendre R vers C , tel que ce sommet est dans $(P \setminus Del) \setminus Q$ (pour qu'il soit effectivement traité par la boucle de la ligne 7). Pour cela, on raisonne par l'absurde : supposons que ce ne soit pas le cas. Alors $((P \setminus Del) \setminus Q) \cap C = \emptyset$, et puisque $ForbidEdges$ n'a aucune arête dans C , alors $C \cap Q = \emptyset$ et on en déduit que $(P \setminus Del) \cap C = \emptyset$. Donc, puisque $C \subseteq R \cup P$ par hypothèse, alors $C \subseteq R \cup Del$. Or, tous les sommets de R et de Del sont des voisins de p , i.e. $R \cup Del \subseteq N_{G_t}(p)$, et donc $C \subseteq N_{G_t}(p)$. Cela implique deux choses : $p \notin C$, et $C \cup \{p\}$ est une clique de G_t . Or, $C \subseteq R \cup Del$ implique que $\mathcal{E}_t(C) = \mathcal{E}_t(C \cup \{p\})$, par définition de Del . Ainsi, $(C \cup \{p\}, [t, \mathcal{E}_t(C)])$ est une clique de L , ce qui contredit le fait que $(C, [t, \mathcal{E}_t(C)])$ est une clique maximale. Par conséquent, $\mathcal{P}_p(k + 1)$ est vraie.

Ainsi, $\mathcal{P}_p(|C|)$ est vraie, et cela montre que C est bien renvoyée par la fonction `GraphCliquesPivot` dans l'algorithme 5.2, ce qui valide cette preuve. □

5.5.2 Complexité en fonction des caractéristiques de l'entrée et de la sortie de l'algorithme

De manière générale, une complexité peut être exprimée en fonction des paramètres d'entrée de l'algorithme, afin de savoir quels sont les paramètres structurels qui déterminent *a priori* l'efficacité du calcul. Elle peut également être exprimée en fonction de la sortie, notamment pour évaluer si l'algorithme est proche ou non d'un calcul optimal, qui serait un calcul dont le nombre d'opérations serait du même ordre de grandeur que la taille de la sortie. Dans cette section, nous nous intéressons à ces deux types de complexité.

Tout au long de cette section, nous nous référerons aux caractéristiques suivantes du flot de liens :

- $m = |E|$: nombre de liens ; notez que chaque lien (b, e, u, v) dans le flot de liens correspond à deux temps $b, e \in T$, donc $|T| \leq 2m$;
- d : degré maximal d'un sommet dans les graphes instantanés G_t ;
- α_T : nombre de cliques maximales en temps ; remarquez que toutes les cliques maximales en temps ne sont pas maximales, et que chaque lien induit une clique maximale en temps, donc $m \leq \alpha_T$;
- α : nombre de cliques maximales ; notez que $\alpha \leq \alpha_T$;
- q : nombre maximal de sommets dans une clique.

Complexité en fonction des paramètres du flot de liens d'entrée

Pour calculer la complexité de l'algorithme 5.1, nous présentons d'abord le lemme 5.3 préliminaire, qui l'exprime en fonction du nombre de cliques maximales en temps α_T . Ensuite, nous estimons une borne supérieure sur α_T pour déduire une expression générale de cette complexité (théorème 5.5). Enfin, nous présentons la complexité en mémoire de l'algorithme (théorème 5.6).

Lemme 5.3. *Les algorithmes sans pivot (5.1) et avec pivot (5.2) s'exécutent en $\mathcal{O}(d^2 \cdot \alpha_T)$.*

Démonstration. Commençons par calculer la complexité de l'algorithme sans pivot (5.1). Rappelons tout d'abord que, d'après le lemme 5.2, pour chaque $t \in T$ il y a exactement une paire (C, \mathcal{N}_C) énumérée dans les ensembles *Cliques* pour chaque clique **maximale en temps** du flot de liens qui commence en t . Il y a donc un total de α_T couples (C, \mathcal{N}_C) énumérés par toute la boucle de la ligne 1.

Premièrement, montrons que la complexité de construction de tous ces couples (C, \mathcal{N}_C) est en $\mathcal{O}(d^2 \cdot \alpha_T)$. Suite à la remarque précédente, on sait qu'il y a au total α_T appels à la fonction **GraphCliques**, en comptant les appels récursifs, car chacun de ces appels renvoie un couple (C, \mathcal{N}_C) . Considérons alors un tel appel et les opérations qui lui sont associées, pour montrer qu'elles ne dépassent jamais $\mathcal{O}(d^2)$:

- le calcul de ses arguments (à la ligne 6 ou à la ligne 15) se fait par l'intersection d'ensembles de taille au plus d , il se fait donc en $\mathcal{O}(d)$;
- les ensembles du couple $(R, P \cup X)$ renvoyé à la ligne 12 sont tous les deux de taille au plus d , donc la sortie de la fonction se fait en $\mathcal{O}(d)$;
- le calcul de Q à la ligne 13 correspond, pour chaque sommet u de P , à l'intersection entre R et les voisins de u dans *ForbidEdges*. Chacun des trois ensembles mentionnés est de taille au plus d , de sorte que ce calcul est effectué en $\mathcal{O}(d^2)$;

- enfin, chaque itération de la boucle de la ligne 14 est constituée d'opérations en temps constant (lignes 16 et 17), et du calcul des arguments et d'un autre appel récursif (ligne 15) dont la complexité est comptabilisée dans celui-ci. Ainsi, les opérations de cette boucle pour l'appel que nous considérons sont en $\mathcal{O}(d)$.

Ainsi, la complexité totale de l'ensemble des appels à **GraphCliques** effectués par l'algorithme 5.1 est en $\mathcal{O}(d^2 \cdot \alpha_t)$.

Deuxièmement, le traitement de chaque paire (C, \mathcal{N}_C) par le test de maximalité de la ligne 9 se fait en $\mathcal{O}(d^2)$, et donc la complexité de tous les tests de maximalité réalisés par la boucle de la ligne 8 se fait en $\mathcal{O}(d^2 \cdot \alpha_T)$. En effet, chaque test nécessite de calculer les temps finaux $\mathcal{E}_t(C)$ et $\mathcal{E}_t(C \cup \{u\})$ pour u dans \mathcal{N}_C . Dans ce qui suit, on montre qu'on peut accéder à ce temps final en $\mathcal{O}(d)$, et comme \mathcal{N}_C contient au plus d sommets, on obtient que le test de maximalité se fait en $\mathcal{O}(d^2)$. La structure de données du flot de liens permet d'accéder directement au temps de fin de chaque lien, de sorte que lors des appels successifs à **GraphCliques**, il est possible de maintenir le temps final de la clique courante R en $\mathcal{O}(d)$. En effet, lorsqu'un sommet u est ajouté à R , à la ligne 15, alors $\mathcal{E}_t(R \cup \{u\}) = \min\{\mathcal{E}_t(R), \min_{v \in R}\{\mathcal{E}_t(u, v)\}\}$, et cette opération peut être effectuée en $\mathcal{O}(|R|) \subseteq \mathcal{O}(d)$. Maintenir $\mathcal{E}_t(C)$ est donc possible en $\mathcal{O}(d)$. Maintenant, $\mathcal{E}_t(C \cup \{u\})$ (ligne 9) est obtenu en ajoutant un sommet à C , donc avec la même procédure, on l'obtient en $\mathcal{O}(|C|) \subseteq \mathcal{O}(d)$. Pour finir, la sortie, ligne 10, a également une complexité en $\mathcal{O}(d)$, et elle est exécutée au plus $|\mathcal{N}_C| \leq d$ fois. Ainsi, la complexité totale du test de maximalité et de la sortie est en $\mathcal{O}(d^2)$, d'où la complexité de la boucle de la ligne 8 en $\mathcal{O}(d^2 \cdot \alpha_T)$.

La complexité globale de l'algorithme 5.1, qui est la somme des deux précédentes, est donc en $\mathcal{O}(d^2 \cdot \alpha_T)$.

Concernant l'algorithme avec pivot (5.2), montrons que sa complexité reste en $\mathcal{O}(d^2 \cdot \alpha_T)$ lorsqu'on ajoute les opérations de **GraphCliquesPivot** à celles de l'algorithme sans pivot. Pour cela, considérons un appel à **GraphCliquesPivot** donné. Fixons $p \in P \cup X$, et étudions le coût qu'ajoute la construction de l'ensemble *Del* par rapport au même appel dans **GraphCliques**. Rappelons que $Del = \{u \in P \cap N_{G_t}(p) \mid \mathcal{E}_t(R \cup \{u, p\}) = \mathcal{E}_t(R \cup \{u\})\}$. Premièrement, d'après le même raisonnement que dans le paragraphe précédent, le calcul de $\mathcal{E}_t(R \cup \{p\})$ se fait en $\mathcal{O}(|R|) \subseteq \mathcal{O}(d)$. Deuxièmement, évaluons le coût du test $\mathcal{E}_t(R \cup \{u, p\}) = \mathcal{E}_t(R \cup \{u\})$, pour chaque $u \in P \cap N_{G_t}(p)$. Fixons donc $u \in P \cap N_{G_t}(p)$. Alors, comme $u \in P$, on a vu que le calcul de $\mathcal{E}_t(R \cup \{u\})$ était déjà pris en compte dans la complexité de l'algorithme sans pivot que l'on vient d'étudier. On peut donc considérer qu'on accède à $\mathcal{E}_t(R \cup \{u\})$ en $\mathcal{O}(1)$. Or, $\mathcal{E}_t(R \cup \{u, p\}) = \min(\mathcal{E}_t(R \cup \{u\}), \mathcal{E}_t(R \cup \{p\}, \mathcal{E}_t(u, p)))$ et donc se calcule également en $\mathcal{O}(1)$. Ainsi, après avoir calculé $\mathcal{E}_t(R \cup \{p\})$ en $\mathcal{O}(d)$, l'ensemble *Del* est construit en $\mathcal{O}(|P \cap N_{G_t}(p)|) \subseteq \mathcal{O}(d)$. Or, on a vu dans la section 5.4.4 que *Del* est calculé pour chaque $p \in P \cup X$, afin de choisir celui qui contient le plus de

sommets à élaguer. Ainsi, Del est calculé $\mathcal{O}(|P \cup X|) \subseteq \mathcal{O}(d)$ fois. On en déduit que le calcul des ensembles Del ajoute au plus des opérations en $\mathcal{O}(d^2)$ par rapport aux appels à **GraphCliqués** dans l'algorithme sans pivot. À cela s'ajoute le calcul de l'ensemble $P \setminus Del$, qui se fait en $\mathcal{O}(d)$ par appel. Or, il y a α_T tels appels. Ainsi, les opérations ajoutées au sein de l'algorithme avec pivot maintiennent la complexité en $\mathcal{O}(d^2 \cdot \alpha_T)$. \square

Nous pouvons maintenant établir la complexité globale de notre algorithme.

Théorème 5.5 (Complexité des algorithmes avec et sans pivot). *La complexité des algorithmes avec pivot (5.1) et sans pivot (5.2) est en $\mathcal{O}(m \cdot 3^{d/3} \cdot 2^q \cdot d^2)$.*

Démonstration. Moon et Moser ont montré que le nombre de cliques maximales dans un graphe à n sommets est en $\mathcal{O}(3^{n/3})$ (borne atteinte dans le pire cas) [MM65]. Concentrons-nous sur l'algorithme sans pivot (5.1). Considérons un appel initial à **GraphCliqués** (ligne 6) : il renvoie un ensemble de paires (C, \mathcal{N}_C) , dont les ensembles C sont des cliques du sous-graphe induit de G_t par les sommets de $N_{G_t}(u) \cap N_{G_t}(v) \cup \{u, v\}$. Ces cliques sont toutes différentes les unes des autres (voir la preuve du lemme 5.2). Comme il y a au plus $d+2$ sommets dans ce sous-graphe induit, alors son nombre de cliques maximales est en $\mathcal{O}(3^{d/3})$. Or, chacune de ces cliques maximales contient au plus q sommets, par définition de q , et elle contient donc au plus 2^q sous-cliques différentes. Par conséquent, ce sous-graphe induit contient au plus $2^q \cdot 3^{d/3}$ cliques différentes, et donc le nombre total de paires (C, \mathcal{N}_C) énumérées par l'appel à **GraphCliqués** considéré est en $\mathcal{O}(2^q \cdot 3^{d/3})$. Enfin, un tel appel est effectué sur chaque lien du flot de liens, de sorte que le nombre total de paires (C, \mathcal{N}_C) énumérées au cours de l'algorithme 5.1 est en $\mathcal{O}(m \cdot 2^q \cdot 3^{d/3})$.

Or, d'après le lemme 5.2, le nombre de paires (C, \mathcal{N}_C) énumérées par l'algorithme 5.1 est exactement α_T . On en déduit que α_T est en $\mathcal{O}(m \cdot 2^q \cdot 3^{d/3})$. Ainsi, d'après le lemme 5.3, la complexité des algorithmes 5.1 et 5.2 est donc en $\mathcal{O}(m \cdot 3^{d/3} \cdot 2^q \cdot d^2)$. \square

Cette complexité est à comparer à celle des algorithmes de l'état de l'art. La complexité de l'algorithme de Viard *et al.* [VML18] est en $\mathcal{O}(2^n \cdot n^2 \cdot m^2 \cdot (n + \log(m)))$ avec $n = |V|$. Celle de Himmel *et al.* [Him+17] est en $\mathcal{O}(m \cdot 3^{c/3} \cdot 2^c \cdot n \cdot |T|)$ avec c la dégénérescence maximale d'un graphe G_t (remarquez que $q - 1 \leq c \leq d$). Enfin, la complexité de l'algorithme de Bentert *et al.* [Ben+19] est en $\mathcal{O}(2^c \cdot \min(m^2, |T|^2) \cdot n^4)$. Bien qu'elles ne soient pas directement comparables à celle du théorème 5.5, ces trois complexités sont toutes des produits d'au moins deux des valeurs n , m et $|T|$ du flot de liens, alors qu'il n'y a qu'un seul facteur m parmi ceux de la nôtre. Notez que toutes ces complexités dépendent également de c , q ou d , mais on sait que ces valeurs sont beaucoup plus faibles que n , m et $|T|$.

lorsqu'on considère des données réelles (voir tableaux 5.1 et 5.2). Ainsi, cette observation suggère que notre algorithme serait plus à même de s'adapter à des flots de liens plus massifs issus de données réelles, ce que nous confirmons par les expériences de la section 5.6.

En ce qui concerne la complexité de l'algorithme avec pivot, il convient de noter que, dans le pire des cas, chaque clique maximale en temps est une clique maximale du flot de liens. Dans ce cas, le pivot ne peut élaguer aucun appel récursif, puisqu'ils renvoient tous une clique maximale. Ainsi, la complexité dans le pire des cas de l'algorithme 5.2 est la même que celle de l'algorithme 5.1. Néanmoins, nous étudions plus en détail l'impact du pivot dans la section suivante, dans laquelle nous exprimons la complexité en fonction de la sortie de l'algorithme.

Enfin, nous montrons ci-dessous que le besoin en mémoire de notre algorithme est proche de la taille m du flot de liens. Il est possible de montrer que les algorithmes proposés par Himmel *et al.* [Him+17] et Bentert *et al.* [Ben+19] ont la même complexité mémoire que le nôtre, tandis que celui de Viard *et al.* [VML18] est exponentiel en le nombre de sommets (il nécessite d'enregistrer l'ensemble des cliques en mémoire vive).

Théorème 5.6 (Complexité en mémoire). *La complexité en mémoire des algorithmes sans pivot (5.1) et avec pivot (5.2) est en $\mathcal{O}(m + q \cdot d)$.*

Démonstration. En pratique, les m liens du flot ont besoin d'être stockés en mémoire. Il faut aussi stocker l'ensemble d'arêtes *ForbidEdges*, qui est au plus de taille m . Remarquez qu'il est possible d'effectuer le test de maximalité de la ligne 9 à l'intérieur de la fonction **GraphCliques** (ou **GraphCliquesPivot**), au moment de décider si la clique est renvoyée ou non, ce qui élimine la nécessité de stocker plus d'une clique en mémoire en même temps. Les cliques peuvent donc être stockées en utilisant un espace mémoire en $\mathcal{O}(q) \subseteq \mathcal{O}(m)$. Enfin, il reste à calculer le coût mémoire de l'arbre des appels récursifs de **GraphCliques** (ou **GraphCliquesPivot**). Chaque appel de la ligne 6 de l'algorithme 5.1 génère une pile d'appels récursifs au plus de taille q . Pour chacun des appels de cette pile, il faut stocker les ensembles R , P , X , Q , ainsi que *Del* dans le cas de l'algorithme avec pivot. Or, chacun de ces ensembles est au plus de taille d . La pile a donc une complexité mémoire en $\mathcal{O}(q \cdot d)$. Toutes ces structures de données s'ajoutent pour donner une complexité mémoire des algorithmes avec et sans pivot en $\mathcal{O}(m + q \cdot d)$. \square

Complexité en fonction des caractéristiques de la sortie de l'énumération

Nous formulons ici la complexité de notre algorithme comme une fonction de α , le nombre de cliques maximales de L , q le nombre maximal de sommets que contiennent les cliques de L , et d le degré maximal des graphes instantanés. Pour ce faire, nous considérons les arbres des appels récursifs de **GraphCliques**, et nous

appliquons le même raisonnement que celui que nous avons utilisé pour exprimer la complexité de l'énumération des cliques maximales dans les graphes, dans le chapitre 4 lors de l'expression de la complexité de l'algorithme d'Eppstein *et al* [ELS10], à la section 4.2.2. Les nœuds internes de ces arbres correspondent aux appels pour lesquels l'ensemble sur lequel itère la boucle de la ligne 14 n'est pas vide (c'est-à-dire qu'elle génère d'autres appels enfants), tandis que les feuilles correspondent aux appels qui n'en génèrent aucun autre. Nous nous concentrerons sur les feuilles de ces arbres des appels, que nous séparons en deux catégories : celles qui produisent une paire (C, \mathcal{N}_C) correspondant à une clique maximale du flot de liens, et celles dont la paire de sortie ne correspond pas à une clique maximale. Ces dernières correspondent à des calculs inutiles, car elles ne contribuent pas à l'énumération des cliques maximales. Comme dans les graphes, une stratégie optimale de pivot couperait ces branches, de sorte à ne laisser que des feuilles qui renvoient une clique maximale. Néanmoins, à la différence des graphes, certaines cliques sont maximales sans être des feuilles de l'arbre des appels, donc le nombre de feuilles qui renvoient une clique maximale n'est pas égal à α .

Notons alors ℓ le nombre total de feuilles de la forêt des arbres des appels, ℓ_{max} le nombre de feuilles qui correspondent à des cliques maximales, et $\ell_{\neg max}$ celles qui ne le sont pas, de sorte que $\ell = \ell_{max} + \ell_{\neg max}$. Nous nous intéressons au rapport suivant :

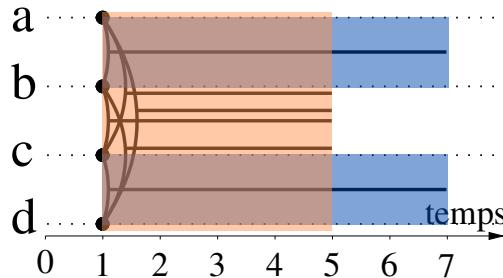
$$r = \frac{\ell_{max}}{\ell} = \frac{\ell_{max}}{\ell_{max} + \ell_{\neg max}}.$$

Ce rapport peut être calculé soit pour l'algorithme 5.1 (sans pivot), soit pour l'algorithme 5.2 (avec pivot). Dans le premier cas, il quantifie l'efficacité maximale possible du pivot : si r est inférieur à 1, cela signifie qu'il y a des appels récursifs qui ne sont pas nécessaires et qui pourraient être élagués par un pivot. Sur un même flot de liens, la comparaison des rapports obtenus par les algorithmes avec et sans pivot montre dans quelle mesure la stratégie du pivot a permis d'élaguer les appels inutiles ou non. Un exemple d'un flot de liens avec le calcul de son rapport r est donné dans la figure 5.6.

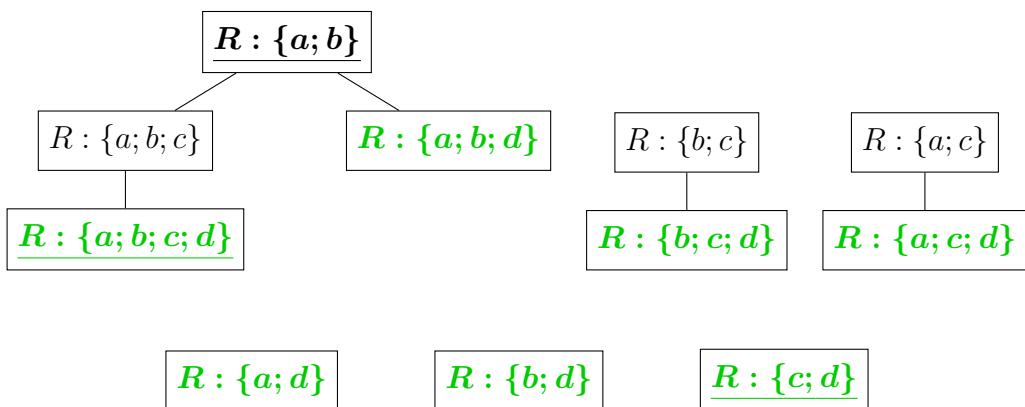
L'utilisation de ce rapport nous permet de décrire la complexité des algorithmes 5.1 et 5.2 en fonction de la sortie, et ainsi de pouvoir évaluer leur différence de performances :

Théorème 5.7 (Complexité en fonction de la sortie). *Avec la définition de r donnée ci-dessus, on a $1 \leq \frac{1}{r} \leq 2^q$, et la complexité des algorithmes 5.1 et 5.2 en fonction de la sortie est en $\mathcal{O}\left(\frac{1}{r} \cdot d^2 \cdot q \cdot \alpha\right)$ (avec le rapport r qui est spécifique à chaque algorithme).*

Démonstration. Montrons d'abord que $1 \leq \frac{1}{r} \leq 2^q$. Il est clair que $\frac{1}{r} \geq 1$. Mainte-



- (a) Flot de liens sur les sommets a, b, c, d , contenant trois cliques maximales : $(\{a, b\}, [1, 7])$, $(\{c, d\}, [1, 7])$ et $(\{a, b, c, d\}, [1, 5])$.



- (b) Ensembles des arbres des appels récursifs à la fonction `GraphCliques` lors du déroulé de l'algorithme 5.1 sur le flot de liens de la figure (a), pour lesquels on représente l'ensemble R associé à chaque appel. Les feuilles des arbres sont colorées en vert, et les cliques R à des maximales du flot de liens sont soulignées.

FIGURE 5.6 – Exemple d'un flot de liens sur lequel on applique l'algorithme 5.1 (sans pivot). La figure (a) représente le flot de liens, et la figure (b) représente les arbres des appels récursifs à la fonction `GraphCliques`. Dans cet exemple, on a $q = 4$, $\alpha = 3$ et $\alpha_T = 11$ (nombre de noeuds des arbres). On compte au total 7 feuilles, en vert ($\ell = 7$), dont les deux qui sont soulignées sont des cliques maximales, ($\ell_{max} = 2$). **Le rapport r vaut donc $\frac{2}{7}$.** Notez qu'une clique peut être maximale sans être une feuille : c'est le cas de $(\{a, b\}, [1, 7])$.

nant, introduisons c_{max} pour représenter le nombre de paires (C, \mathcal{N}_C) qui sont énumérées par **GraphCliques** telles que C est une clique maximale du graphe instantané associé G_t . Par exemple, dans la figure 5.6, $c_{max} = 1$, pour la clique $\{a, b, c, d\}$. Ces paires ne peuvent être énumérées que par des feuilles de l'arbre des appels, car si C est une clique maximale du graphe G_t , on ne peut pas lui ajouter de sommet pour en faire une clique plus grosse. En outre, la clique maximale en temps associée à chacune est nécessairement maximale en sommets, et donc maximale. Ainsi, c_{max} vérifie $c_{max} \leq \ell_{max}$. De plus, chaque clique maximale d'un graphe G_t contient au plus 2^q sous-cliques, donc il y a au plus $2^q \cdot c_{max}$ paires (C, \mathcal{N}_C) énumérées au total par l'algorithme. Enfin, chaque feuille de l'arbre des appels renvoie une paire, donc on a $\ell \leq 2^q \cdot c_{max}$, et donc $\frac{1}{r} \leq \frac{2^q \cdot c_{max}}{\ell_{max}} \leq 2^q$.

Exprimons à présent l'expression de la complexité. Par définition de q , nous savons que la profondeur d'un arbre des appels de **GraphCliques** est au plus q . Il y a donc au plus $q \cdot \ell$ noeuds dans toute la forêt des arbres des appels. Or nous avons vu dans la preuve du lemme 5.3 qu'il y a exactement un de ces noeuds par clique maximale en temps. Il y a donc α_T noeuds, ce qui implique que $\alpha_T \leq q \cdot \ell$. Ainsi, en utilisant l'expression de la complexité du lemme 5.3, on obtient que la complexité de l'algorithme 5.1 (et de l'algorithme 5.2) est en $\mathcal{O}(d^2 \cdot q \cdot \ell)$. Or, comme $\ell = \frac{1}{r} \cdot \ell_{max}$, et que $\ell_{max} \leq \alpha$, alors la complexité est en $\mathcal{O}\left(d^2 \cdot q \cdot \frac{1}{r} \cdot \alpha\right)$. \square

Comme on l'a vu lorsqu'on a introduit le rapport dans les graphes, à la section 4.2.2 du chapitre 4, les bornes de $\frac{1}{r}$ données par le théorème 5.7 sont précises. Pour illustrer ceci, nous pouvons prendre les mêmes exemples que sur les graphes, en considérant l'exemple d'un flot de liens qui est équivalent à un graphe, car tous ses liens ont le même intervalle d'existence. Pour rappel, nous avons montré que dans le cas d'un graphe qui correspond à une clique, le rapport $\frac{1}{r}$ est égal à 1 pour l'algorithme avec pivot, et il est égal à 2^{q-2} pour l'algorithme sans pivot.

Néanmoins, nous verrons dans la section 5.6 que $\frac{1}{r}$ est petit dans les expériences avec pivot : il est inférieur à 2 dans toutes les expériences sauf une. Cela signifie qu'il n'a pas un comportement exponentiel sur les flots de liens réels que nous étudions, contrairement à sa borne supérieure théorique. D'après cette observation et la complexité calculée, nous pouvons affirmer que le temps d'exécution de notre algorithme avec pivot est proche du meilleur que l'on puisse attendre. En effet, puisque le temps d'exécution doit traiter tous les sommets de toutes les α_T cliques maximales, et que la taille des plus grandes cliques est de q , alors on s'attend à ce qu'il y ait nécessairement un facteur $q \cdot \alpha$ dans la complexité. Notre algorithme n'y ajoute qu'un facteur multiplicatif $\frac{1}{r} \cdot d^2$, qui peut être élevé en pratique, mais nous montrons dans la section suivante ses bonnes performances sur les graphes issus du monde réels. En effet, dans ces graphes, bien que d puisse être élevé, les degrés moyens sont faibles en pratique, et la taille des voisinages bornée par d est en fait très éloignée de d dans la plupart des opérations réalisées par l'algorithme.

5.6 Évaluation expérimentale

Dans cette section, nous faisons une étude expérimentale des algorithmes 5.1 et 5.2, pour montrer leurs performances en pratique. Pour cela, nous comparons les implémentations des algorithmes présentés dans la section 5.4 à celles fournies dans la littérature par Viard *et al.* [VML18]², Himmel *et al.* [Him+17]³, et Bentert *et al.* [Ben+19]⁴. Ensuite, nous montrons que notre algorithme permet de traiter des flots de liens massifs issus du monde réel de plusieurs dizaines de millions de liens, puis nous étudions l'impact qu'a le pivot sur les temps de calcul, et enfin nous présentons les résultats d'une implémentation parallèle.

5.6.1 Dispositif expérimental

Machine. Nous avons réalisé les expériences sur une machine équipée de 2 processeurs Intel Xeon Silver 4216 avec 32 coeurs chacun et 380 Go de RAM.

Implémentations. Nous avons réalisé deux implémentations des algorithmes 5.1 et 5.2 : l'une en Python et l'autre en C++. Nous utilisons l'implémentation Python pour nous comparer aux méthodes de l'état de l'art qui sont codées en Python, et l'implémentation C++ pour passer à l'échelle à des flots de liens plus massifs. L'implémentation en C++ est inspirée de l'implémentation efficace de l'algorithme de Bron-Kerbosch faite par Eppstein *et al.* [ELS10], qui énumère les cliques maximales dans les graphes. Comme l'utilisation d'un pivot réduit le temps d'exécution, nous utilisons par défaut l'algorithme 5.2, sauf dans la section 5.6.4 dans laquelle nous analysons l'efficacité de la stratégie de pivot en comparant les exécutions des algorithmes 5.1 et 5.2.

Données et prétraitement. En suivant les travaux de Viard *et al.* [VML18], nous utilisons des jeux de données qui correspondent à des flots de liens *instantanés*, dans lesquels tous les liens ont une durée égale à 0, et nous associons une durée Δ à chaque lien (t, u, v) en le remplaçant par $(t, t + \Delta, u, v)$ (notez que cela fusionne tous les liens entre des mêmes sommets séparés par un intervalle de durée inférieure à Δ). Cela nous permet d'étudier le comportement de notre algorithme avec des entrées variées, tout en permettant de le comparer à ceux qui étudient les Δ -cliques dans les flots de liens instantanés, puisque Viard *et al.* [VML18] ont montré que les deux problèmes sont équivalents. Nous ne nous intéressons ici qu'au comportement de notre code, et pas à l'analyse des données qu'il permettrait de développer.

Nous utilisons deux familles différentes de jeux de données. La première, dont les principales caractéristiques sont détaillées dans le tableau 5.1, correspond aux

2. https://bitbucket.org/tiph_viard/cliques
 3. <https://fpt.akt.tu-berlin.de/temporalcliques>
 4. <https://fpt.akt.tu-berlin.de/temporalkplex>

jeux de données utilisés par Bentert *et al.* [Ben+19] pour comparer leur propre algorithme à la littérature. On y choisit des valeurs de Δ identiques à celles utilisées par Bentert *et al.* [Ben+19], à des fins de comparaison. Notez que la valeur de Δ influence le nombre de liens (lorsque Δ augmente, plusieurs liens sur des mêmes sommets peuvent se chevaucher dans le temps, et donc fusionner en un seul lien), ainsi que sur les autres paramètres du tableau (sauf le nombre de sommets). La seconde famille correspond à des jeux de données plus massifs, dont les caractéristiques sont données dans le tableau 5.2. Nous utilisons ces jeux de données pour montrer que notre algorithme passe à l'échelle de flots de liens allant jusqu'à plusieurs dizaines de millions de liens. On y choisit des valeurs de Δ en fonctions de Θ , la durée totale du flot de liens : 0, $\Theta/10\,000$, et $\Theta/100$. Les résultats des expériences correspondantes sont détaillés respectivement dans la section 5.6.2 et la section 5.6.3.

5.6.2 Comparaison à l'état de l'art

Les implémentations de nos algorithmes sont beaucoup plus efficaces en temps de calcul que celles de l'état de l'art. Pour l'illustrer, nous présentons dans le tableau 5.3 les temps de calcul (en secondes) pour les jeux de données du tableau 5.1, qui sont les jeux de données étudiés par Bentert *et al.* [Ben+19]. Comme les algorithmes de l'état de l'art sont implémentés en **Python**, nous les comparons d'abord avec notre implémentation en **Python**, afin d'évaluer l'amélioration apportée par notre algorithme. Nous observons que notre algorithme est le plus rapide dans tous les cas testés. Parmi les trois algorithmes de l'état de l'art, **HMNS** [Him+17] est clairement le moins compétitif, tandis que les algorithmes **BHM+** [Ben+19] et **VML** [VML18] donnent des résultats comparables en termes de temps d'exécution : selon le flot de liens, l'un ou l'autre est plus rapide, tout en restant dans le même ordre de grandeur. Par ailleurs, nous observons également que notre implémentation en **C++** est très efficace sur ces jeux de données : dans la plupart des cas, le temps de calcul est inférieur à 1 seconde, et ne dépasse jamais 3 secondes, même sur le jeu de données *Flights*, pour lequel il est plus de 100 fois plus rapide que toutes les autres implémentations (y compris notre implémentation en **Python**).

Pour mieux visualiser les différences de performance, la figure 5.7 montre les temps de calcul en secondes des différentes méthodes pour tous les flots de liens de nos jeux de données. Les différents flots de liens sont classés sur l'axe des abscisses par nombre croissant de liens m ; notez que les échelles sont logarithmiques. Une expérience est faite par valeur de Δ pour chacun des jeux de données, et chacune correspond à trois temps de calcul : le meilleur temps d'exécution de l'état de l'art, le temps d'exécution de notre implémentation **Python**, et celui de notre implémentation **C++**. Les trois lignes horizontales en haut de la figure correspondent aux exécutions pour lesquelles le calcul a été interrompu, parce qu'il a dépassé 24 heures ou 380 Go de RAM : un point sur l'une de ces lignes signifie que le calcul du jeu de données et de l'algorithme correspondant n'a pas terminé, pour l'une de ces deux raisons. Nous

Flot de liens	Δ	n	m	$ T $	d	α	q
<i>DNC [Kun13]</i>	0		8 111	5 394	72	8 111	2
	125	1 866	7 967	5 291	99	8 038	4
	3 125		7 454	4 945	108	7 834	4
<i>Hypertext [Ise+11]</i>	0		20 818	5 246	9	19 037	6
	125	113	6 323	2 918	14	6 859	7
	3 125		4 082	2 077	48	6 308	7
<i>Highschool-2011 [FB14a]</i>	0		28 539	5 609	8	26 384	5
	125	126	6 472	2 808	19	7 732	7
	3 125		3 636	1 481	34	7 500	10
<i>Hospital-Ward [Van+13]</i>	0		32 424	9 453	7	27 835	5
	125	75	7 971	4 799	12	9 731	6
	3 125		3 033	2 379	25	9 856	9
<i>Highschool-2012 [FB14a]</i>	0		45 047	11 273	5	42 105	5
	125	180	11 329	5 972	10	12 115	5
	3 125		5 691	2 886	18	7 268	7
<i>Facebook-like [POC09b]</i>	0		59 795	58 911	31	59 795	2
	125	1 899	50 056	49 434	78	50 080	3
	3 125		34 116	33 801	92	34 342	4
<i>AS-733 [LKF05]</i>	0		110 581	725	1 458	97 687	10
	125	7 716	32 485	528	1 568	41 965	12
	3 125		21 466	485	1 851	49 293	18
<i>Primary-School [GBC14]</i>	0		125 773	3 100	4	106 879	5
	125	242	49 530	3 081	16	67 820	6
	3 125		19 513	2 924	50	194 231	14
<i>Highschool-2013 [MFB15]</i>	0		188 508	7 375	4	172 035	5
	125	327	36 277	6 942	14	41 534	6
	3 125		15 764	4 054	30	28 357	8
<i>London [WM16]</i>	0		312 164	1 238	7	294 269	3
	125	270	27 595	1 236	7	28 683	3
	3 125		768	129	7	778	3
<i>Paris [WM16]</i>	0		353 226	1 196	8	342 540	3
	125	302	50 248	1 196	8	51 084	3
	3 125		1 080	54	8	1 093	3
<i>Flights [WM16]</i>	0		415 000	480	134	229 144	19
	125	299	415 000	480	134	229 144	19
	3 125		37 862	480	139	368 756	19
<i>Infectious [Ise+11]</i>	0		415 912	76 944	4	338 815	5
	125	10 972	100 329	39 435	15	138 670	7
	3 125		44 767	21 866	43	150 883	16
<i>New-York [WM16]</i>	0		468 897	1 440	12	442 341	3
	125	417	113 651	1 440	12	117 481	3
	3 125		661	201	12	674	3

TABLEAU 5.1 – Caractéristiques des flots de liens étudiés par Bentert *et al.* [Ben+19]. Δ (en secondes) correspond à la durée de chaque lien, m au nombre de liens, d au degré maximal, α au nombre de cliques maximales, et q au nombre maximal de nœuds d'une clique. Les valeurs de Δ sont choisies en fonctions de Θ , la durée totale du flot de liens : 0, $\Theta/10\ 000$, et $\Theta/100$.

Flot de liens	Δ	n	m	$ T $	d	α	q
<i>StackExchange [PBL17]</i>	0	194 085	1 108 715	1 105 102	3	1 108 715	2
	23 961		870 128	867 953	47	894 317	5
	2 396 149		751 974	750 285	671	1 030 023	10
<i>WikiTalk [PBL17]</i>	0	1 140 149	6 092 321	5 799 206	22	6 092 321	2
	20 048		4 123 960	3 987 802	12 205	4 207 362	7
	2 004 838		3 078 861	2 999 169	29 543	4 500 754	10
<i>YouTube [Mis09]</i>	0	3 223 643	12 223 774	203	28 714	12 253 571	17
	1 944		12 223 774	203	28 714	12 253 571	17
	194 400		10 310 419	203	28 714	10 656 065	17
<i>Copresence-Thiers [RA15]</i>	0	328	18 613 039	8 938	79	131 251	80
	38		3 857 645	8 928	100	412 553	80
	3 804		147 125	5 830	217	5 572 145	102
<i>Wikipedia [Mis09]</i>	0	1 870 709	39 949 279	12 098 254	3 463	39 935 611	13
	19 318		39 246 821	11 850 400	7 216	40 898 684	28
	1 931 869		38 737 308	11 702 028	36 121	45 440 988	28
<i>StackOverflow [PBL17]</i>	0	2 601 977	47 902 566	41 484 769	4	47 902 566	2
	23 970		33 948 538	30 678 295	90	35 298 283	6
	2 397 055		29 583 489	27 055 274	1 443	43 989 692	13
<i>Soc-Bitcoin-bitcoin [RA15]</i>	0	24 575 382	122 378 012	19 444 110	3 769	119 172 078	219
	15 653		93 897 987	15 225 966	28 144	-	-
	1 565 366		86 668 193	13 943 567	170 760	-	-

TABLEAU 5.2 – Caractéristiques des flots de liens massifs utilisés pour étudier le passage à l'échelle de l'algorithme. Les valeurs de Δ (en secondes) sont fixées à 0, $\Theta/10\ 000$ et $\Theta/100$, où Θ est la durée totale du flot de liens. Le symbole “-” signifie qu'aucune des implémentations disponibles ne permet d'énumérer l'ensemble cliques maximales.

observons trois couches distinctes de points, ce qui montre que l'implémentation C++ est plus efficace que l'implémentation Python, elle-même plus efficace que l'état de l'art. Enfin, nous remarquons une tendance à avoir un gain légèrement plus élevé avec des flots de liens plus massifs.

Ensuite, pour mieux évaluer le gain en temps de calcul de notre algorithme par rapport à l'état de l'art, la figure 5.8 affiche le facteur d'accélération réalisé par nos deux implémentations par rapport à l'algorithme le plus efficace de l'état de l'art, dans tous les cas où au moins un algorithme de l'état de l'art est capable de fournir un résultat en moins de 24 heures et 380 Go de RAM. Ce facteur correspond au rapport du temps d'exécution de l'état de l'art divisé par le temps de notre implémentation, Python ou C++. La ligne horizontale tracée en bas montre que ce facteur est toujours supérieur à 1. Ainsi, les implémentations de Python et de C++ sont toutes deux plus efficaces que les autres implémentations de l'état de l'art, dans l'ensemble des expériences réalisées. De plus, dans la plupart des expériences, ce facteur d'accélération est supérieur à 10, même pour l'implémentation Python. En ce qui concerne l'implémentation C++, ce facteur est toujours supérieur à 10 sauf pour les 3 plus petits flots de liens, et il peut atteindre jusqu'à 10^4 pour les plus grands flots de liens.

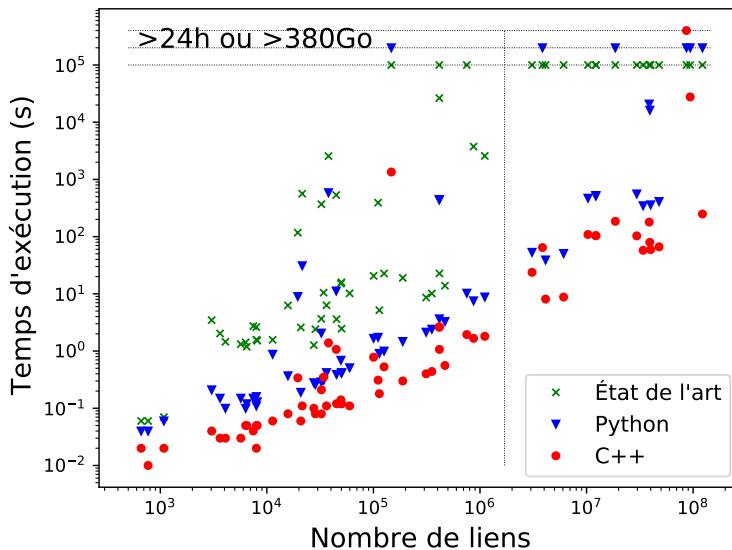


FIGURE 5.7 – Résumé des temps de calcul des énumérations des cliques maximales en fonction du nombre de liens m , pour tous les flots de liens des tableaux 5.1 et 5.2. L'état de l'art correspond au meilleur temps réalisé par l'une des trois méthodes de l'état de l'art. Python et C++ correspondent à nos implémentations. Les trois lignes du haut représentent les énumérations qui ont été interrompues parce qu'elles dépassent 24 heures ou 380 Go de RAM. La ligne verticale à 10^6 liens illustre le passage à l'échelle de notre algorithme sur les deux derniers ordres de grandeur.

Dataset	Δ	C++	Python	BHM+	VML	HMNS
<i>DNC</i>	0	0,05	0,13	24	1,5	178
	125	0,05	0,11	24	2,6	155
	3 125	0,04	0,15	23	2,7	80
<i>Hypertext</i>	0	0,06	0,19	2,6	2,8	65
	125	0,05	0,10	1,4	1,8	6,9
	3 125	0,03	0,10	1,4	2,0	3,6
<i>Highschool-2011</i>	0	0,08	0,26	2,4	2,6	95
	125	0,05	0,12	1,2	1,6	5,4
	3 125	0,03	0,15	2,0	4,1	3,1
<i>Hospital-Ward</i>	0	0,08	0,29	3,7	4,3	271
	125	0,02	0,16	1,6	2,6	17
	3 125	0,04	0,21	3,5	7,8	5,0
<i>Highschool-2012</i>	0	0,12	0,39	3,6	4,2	183
	125	0,06	0,88	1,6	2,1	12
	3 125	0,03	0,15	1,3	1,5	3,8
<i>Facebook-like</i>	0	0,11	0,51	27	10	129
	125	0,12	0,42	26	15	96
	3 125	0,35	0,31	25	10	59
<i>AS-733</i>	0	0,31	1,7	569	392	-
	125	0,21	2,1	368	1 581	528
	3 125	0,11	31	562	\times	572
<i>Primary-School</i>	0	0,53	1,0	22	30	716
	125	0,14	0,69	15	37	125
	3 125	0,34	9,0	204	854	117
<i>Highschool-2013</i>	0	0,30	1,5	19	24	1 420
	125	0,11	0,42	6,3	11	54
	3 125	0,08	0,37	6,3	11	14
<i>London</i>	0	0,40	2,1	8,6	11	3 712
	125	0,10	0,28	2,1	1,3	45
	3 125	0,01	0,04	1,5	0,06	1,6
<i>Paris</i>	0	0,44	2,4	10	13	4 260
	125	0,12	0,41	2,8	2,5	101
	3 125	0,02	0,06	1,7	0,07	1,8
<i>Infectious</i>	0	1,1	3,7	665	22	1 206
	125	0,78	1,7	634	20	945
	3 125	1,1	11	818	534	1 004
<i>Flights</i>	0	2,7	443	36 859	\times	26 109
	125	2,6	442	37 076	\times	26 411
	3 125	1,4	585	13 420	\times	2 555
<i>New-York</i>	0	0,56	3,3	13	18	6 084
	125	0,18	0,91	5,2	6,9	304
	3 125	0,02	0,04	2,3	0,06	2,5

TABLEAU 5.3 – Comparaison des temps de calcul (en secondes) de nos implémentations **C++** et **Python**, et des implémentations de l’état de l’art, **BHM+** [Ben+19], **VML** [VML18] et **HMNS** [Him+17], sur les flots de liens utilisés par Bentert *et al.* [Ben+19] et décrits dans le tableau 5.1. Le symbole “-” signifie que le temps de calcul dépasse 24 heures, et le symbole “ \times ” signifie que la mémoire nécessaire au calcul dépasse 380 Go de RAM.

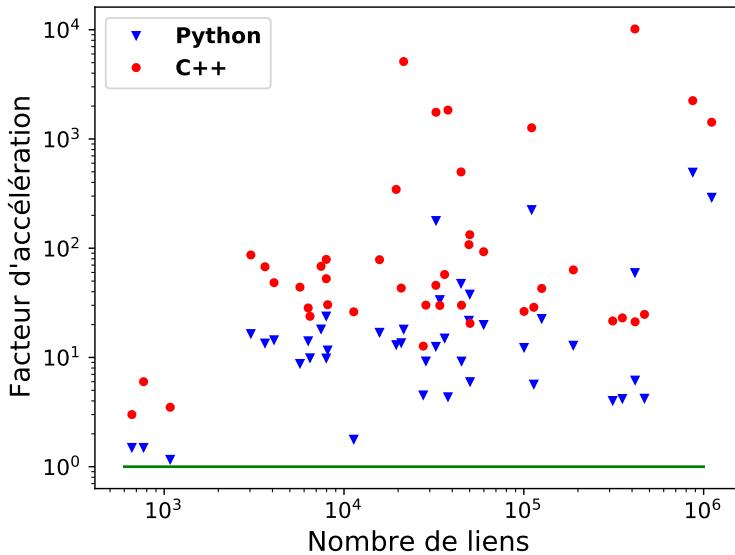


FIGURE 5.8 – Facteur d'accélération de l'énumération, en fonction du nombre de liens m . Il s'agit du meilleurs temps d'exécution des implémentations de l'état de l'art divisé par le temps d'exécution de nos implémentations Python et C++. Il y a un point par jeu de données pour lequel au moins un algorithme de l'état de l'art termine en moins de 24 heures et en utilisant moins de 380 Go de RAM.

5.6.3 Passage à l'échelle sur des flots de liens massifs issus du monde réel

Nous avons vu dans la section précédente que notre algorithme est plus efficace que les implémentations actuellement disponibles dans l'état de l'art. Il est donc pertinent de voir dans quelle mesure il peut être utilisé sur des jeux de données plus massifs. Dans le tableau 5.4, nous présentons les temps d'exécution des énumérations de cliques maximales effectuées sur les flots de liens massifs décrits par le tableau 5.2. Un symbole “-” signifie que l'énumération ne se termine pas en moins de 24 heures pour l'algorithme examiné, et le symbole “×” indique qu'elle nécessite plus de 380 Go de RAM. Dans la figure 5.7, nous avons placé une ligne verticale pour montrer la limite au-delà de laquelle l'état de l'art ne fournit pas de résultat.

On observe que les implémentations de l'état de l'art ne permettent pas d'énumérer les cliques maximales sur ces jeux de données, à l'exception de **VML** [VML18] sur le jeu de données *StackExchange* avec $\Delta = 0s$ et $\Delta = 23\,961s$, et dans ce cas, il est beaucoup moins efficace que notre implémentation **Python**. Dans tous les autres cas, les expériences ont été interrompues soit pour des raisons de temps, soit pour des raisons de consommation de mémoire. L'implémentation **Python** de notre algorithme permet d'énumérer les cliques maximales pour 15 des 21 expériences considérées. Enfin, nous observons un gain significatif de temps de calcul de l'implémentation **C++** par rapport à l'implémentation **Python**. Elle permet d'énumérer

Dataset	Δ	C++	Python	BHM+	VML	HMNS
<i>StackExchange</i>	0	1,8	8,8	-	2 576	-
	23 961	1,7	7,5	-	3 747	-
	2 396 149	1,9	10	-	×	-
<i>WikiTalk</i>	0	8,8	50	-	×	-
	20 048	8,1	39	-	×	-
	2 004 838	23	53	-	×	-
<i>YouTube</i>	0	103	522	-	×	-
	1 944	104	513	-	×	-
	194 400	109	470	-	×	-
<i>Copresence-Thiers</i>	0	185	-	-	×	-
	38	64	-	-	×	-
	3 804	1 347	-	-	×	-
<i>Wikipedia</i>	0	59	359	-	×	-
	19 318	79	16 223	-	×	-
	1 931 869	179	20 767	-	×	-
<i>StackOverflow</i>	0	66	410	-	×	-
	23 970	57	347	-	×	-
	2 397 055	103	557	-	×	-
<i>Soc-Bitcoin</i>	0	249	-	-	×	-
	15 653	27 660	-	-	×	-
	1 565 366	-	-	-	×	-

TABLEAU 5.4 – Comparaison des temps de calcul (en secondes) de nos implémentations **C++** et **Python** et des implémentations de l’état de l’art, **BHM+** [Ben+19], **VML** [VML18] et **HMNS** [Him+17], sur les flots de liens massifs décrits dans le tableau 5.2. Un symbole “-” signifie que le temps de calcul dépasse 24 heures, et un symbole “×” signifie que la mémoire nécessaire au calcul dépasse 380 Go de RAM.

les cliques maximales dans des flots de liens sur lesquels il n'était pas possible de le faire auparavant, car elle termine pour toutes les expériences, à l'exception du flot de liens *Soc-Bitcoin* avec une durée de lien $\Delta = 1\ 565\ 366s$.

5.6.4 Mesure de l'efficacité du pivot

Comme on l'a vu précédemment, l'introduction du pivot n'apporte pas de gain théorique par rapport à la complexité de l'algorithme. Néanmoins, il est connu que pour l'énumération des cliques maximales dans les graphes par l'algorithme de Bron-Kerbosch, il réduit le temps d'exécution.

Nous analysons et commentons donc ici l'intérêt de l'utilisation de notre pivot, tel que présenté dans l'algorithme 5.2, en évaluant les gains en temps de calcul qu'il permet. Pour cela, on s'appuie sur la complexité de l'algorithme 5.1 en fonction de la sortie de l'énumération, qui est en $\mathcal{O}\left(\frac{1}{r} \cdot d^2 \cdot q \cdot \alpha\right)$ d'après le théorème 5.7, où d est le degré maximal dans un graphe instantané G_t , q est la taille maximale d'une clique, α est le nombre de cliques maximales et r est le rapport des feuilles des arbres des appels à `GraphCliques` (pour l'algorithme sans pivot) ou `GraphCliquesPivot` (pour l'algorithme avec pivot) qui correspondent à des cliques maximales du flot de liens. Dans l'énumération sans pivot, r permet de quantifier le gain potentiel que le pivot peut apporter ; par comparaison avec le cas avec pivot, r quantifie l'efficacité réelle du pivot : plus r se rapproche de 1, moins il reste de branches inutiles dans les arbres des appels.

Le tableau 5.5 donne les temps de calcul de l'implémentation en C++, avec et sans pivot, sur les flots de liens massifs du tableau 5.2. Dans les deuxièmes et troisièmes colonnes, qui correspondent respectivement au calcul avec et sans pivot, nous indiquons les temps d'énumération en secondes, le rapport r rappelé ci-dessus et défini dans la section 5.5.2, ainsi que le facteur $1/r$, tel qu'il apparaît dans l'expression de complexité du théorème 5.7, rappelée ci-dessus.

Dans ce tableau, nous observons tout d'abord que le pivot permet de réaliser l'énumération des cliques maximales plus rapidement, dans tous les cas sauf pour les flots de liens *StackExchange* et *WikiTalk*. On peut même constater que dans 5 cas, la version avec pivot termine dans nos limites de temps et de mémoire alors que la version sans pivot ne termine pas. En fait, le facteur d'accélération entre les algorithmes sans et avec pivot peut varier beaucoup d'un jeu de données à l'autre, et il n'est pas directement proportionnel à l'amélioration du facteur $\frac{1}{r}$. Cela provient du coût du calcul du pivot : on a vu que le calcul du pivot au sein d'un appel se faisait en $\mathcal{O}(d^2)$, ce qui ralentit l'algorithme. Néanmoins, ces résultats montrent que l'élagage des branches permet de gagner plus de temps que ce qui est perdu avec le calcul du pivot, à l'exception deux cas cités ci-dessus. Dans ces cas, le rapport r est déjà presque parfait pour l'algorithme sans pivot, et donc l'ajout du pivot ne fait que perdre du temps par son calcul. Enfin, pour le flot de liens *StackOverflow*, on

Link stream		With pivot			Without pivot		
Dataset	Δ	t	r	$1/r$	t	r	$1/r$
<i>StackExchange</i>	0	1,8	1,000	1	1,8	1,000	1
	23 961	1,7	1,000	1	1,7	1,000	1
	2 396 149	1,9	0,997	1,003	1,7	0,928	1,078
<i>WikiTalk</i>	0	8,8	1,000	1	7,7	1,000	1
	20 048	8,1	1,000	1	7,7	1,000	1
	2 004 838	23	0,995	1,005	23	0,891	1,122
<i>YouTube</i>	0	103	0,907	1,103	135	0,332	3,012
	1 944	104	0,907	1,103	135	0,332	3,012
	194 400	109	0,900	1,111	137	0,298	3,356
<i>Copresence-Thiers</i>	0	185	0,087	11,49	-	-	-
	38	64	0,757	1,321	-	-	-
	3 804	1 347	0,854	1,171	-	-	-
<i>Wikipedia</i>	0	59	1,000	1	78	0,987	1,013
	19 318	79	1,000	1	1 143	0,043	23,26
	1 931 869	179	0,999	1,001	1 618	0,034	29,41
<i>StackOverflow</i>	0	66	1,000	1	90	1,000	1
	23 970	57	1,000	1	73	1,000	1
	2 397 055	103	0,996	1,004	122	0,881	1,135
<i>Soc-Bitcoin</i>	0	249	0,972	1,029	-	-	-
	15 653	27 660	0,911	1,098	-	-	-
	15 653 366	-	-	-	-	-	-

TABLEAU 5.5 – Comparaison des temps de calcul, en secondes, entre l’implémentation en C++ qui utilise un pivot et celle sans pivot. Le facteur r , défini dans la Section 5.5.2, est égal au rapport des feuilles des arbres des appels de `GraphCliques` qui correspondent à une clique maximale du flot de liens. Le facteur $\frac{1}{r}$ est présent dans la formule de complexité du théorème 5.7. On voit qu’il reste faible en pratique, malgré son caractère exponentiel théorique.

constate que le rapport vaut déjà 1 dans l'algorithme sans pivot, mais que le pivot introduit quand même un gain dans le temps de l'énumération. Sans en connaître la raison exacte, nous supposons que le pivot permet dans ce cas de réduire l'arbre des appels, en le réorganisant de telle manière qu'il supprime des noeuds internes inutiles, ce qu'on n'est pas en capacité de mesurer avec le rapport r .

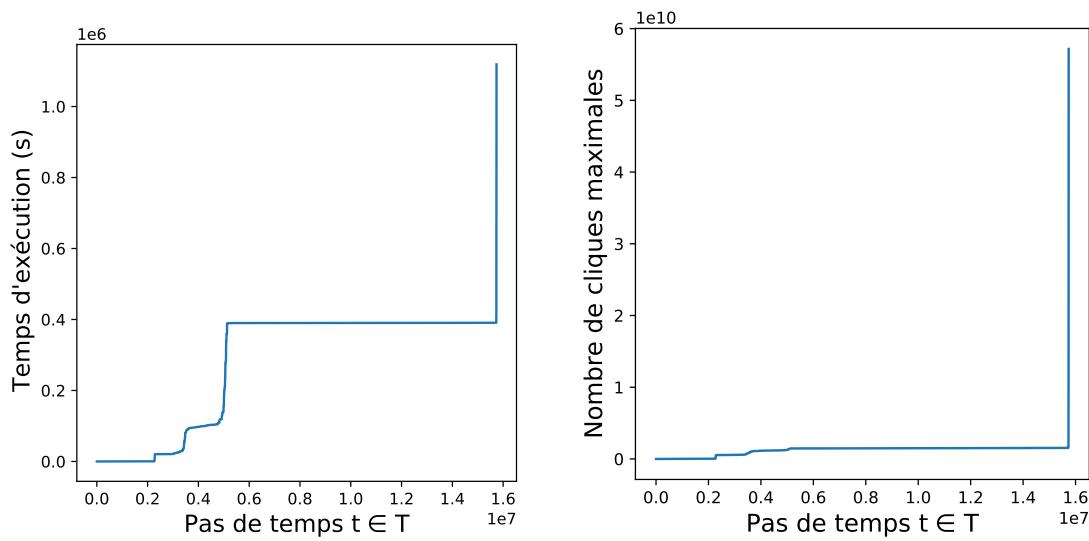
Le rapport r est supérieur à 0.9 dans 18 expériences sur 21 avec le pivot, alors que ce n'est pas le cas dans la plupart des expériences sans pivot. Cela indique que le pivot permet de couper presque toutes les branches inutiles des appels récursifs, sauf dans le cas du jeu de données *Copresence-Thiers*. On remarque que ce jeu de données avec une durée des liens $\Delta = 0s$ se démarque clairement, car il n'y a que 8,7% des feuilles des arbres des appels de **GraphCliques** qui sont des cliques utiles à l'énumération.

On note aussi que le facteur $1/r$, qui peut être très grand en théorie puisqu'il est en $\mathcal{O}(2^q)$ d'après le théorème 5.7, reste en fait relativement petit en pratique. Avec un pivot, ce facteur ne dépasse jamais 2, sauf dans le cas de *Copresence-Thiers* avec $\Delta = 0s$ mentionné ci-dessus. Ainsi, d'après cette observation, la complexité avec pivot peut souvent être considérée en pratique comme étant de l'ordre de $\mathcal{O}(d^2 \cdot q \cdot \alpha)$, ce qui correspond à un facteur d^2 de la taille de la sortie (en $\mathcal{O}(q \cdot \alpha)$), et montre donc que la complexité de l'énumération est proche de l'optimal, dans le cas de flots de liens issus du monde réel, qui sont peu denses.

Enfin, même avec un pivot, le calcul ne termine pas pour la plus grande valeur de Δ sur le jeu de données *Soc-Bitcoin*, dans les limites de notre dispositif expérimental. Dans la figure 5.9, on présente les caractéristiques de l'énumération des cliques maximales dans ce flot de liens, qu'on a laissé tourner près de deux semaines avant de stopper le calcul.

La figure 5.9 (a) représente la distribution cumulative du temps de calcul de l'énumération des cliques maximales par l'algorithme avec pivot, et la figure 5.9 (b) la distribution cumulative du nombre de cliques maximales énumérées, en fonction du temps $t \in T$ du flot de liens. On y constate que la quasi-intégralité du temps de calcul est consommée par seulement deux instants du flot de liens, et notamment que le dernier est bloquant : il a consommé à lui tout seul près d'une semaine de calcul, avant d'être interrompu. Pour expliquer ce goulot d'étranglement dans l'énumération, la figure 5.9 (b) présente le nombre de cliques maximales énumérées de manière cumulative par l'algorithme au cours du temps T du flot de liens. On remarque que quasiment toutes les cliques maximales énumérées le sont au niveau du dernier instant $t \in T$ qui a pu être traité par l'algorithme avant d'être interrompu. Ici, la cause du blocage est donc la densité du graphe G_t associé qui génère trop de cliques pour qu'elles puissent être énumérées en temps raisonnable.

Il existe d'autres jeux de données plus importants sur lesquels aucune méthode n'est capable de terminer dans le temps et la mémoire alloués, comme le flot de liens



(a) Distribution cumulative du temps d'exécution de l'algorithme avec pivot, en secondes, en fonction du temps $t \in T$ du flot de liens.

(b) Distribution cumulative du nombre de cliques maximales énumérées, en fonction du temps $t \in T$ du flot de liens.

FIGURE 5.9 – Étude de la répartition du temps de calcul et du nombre de cliques énumérées au bout de deux semaines pour le flot de liens *soc bitcoin* avec $\Delta = \Theta/100$, pour lequel le calcul ne termine pas dans cette limite de temps pour l'algorithme avec pivot. Les distributions sont cumulatives, et affichées en fonction du temps $t \in T$ du flot de liens.

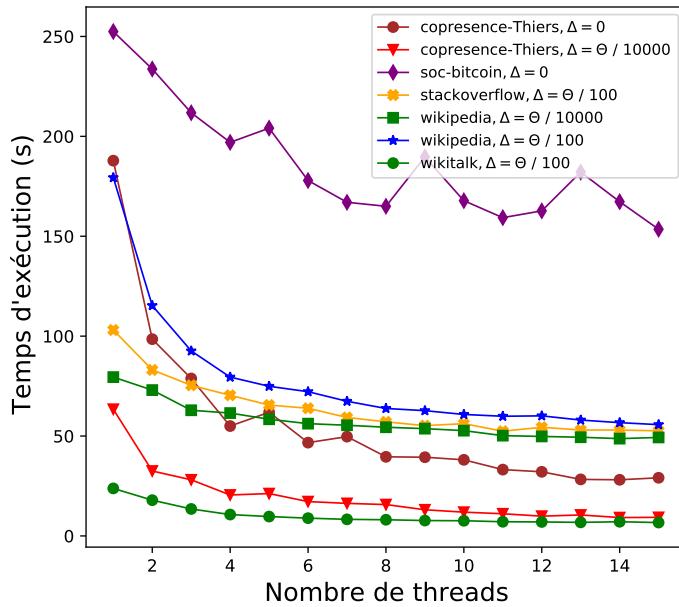
Flickr [Mis+08], dont les graphes instantanés sont très denses : l'algorithme met un temps rédhibitoire à traiter le premier instant, comme ce que nous venons de décrire avec *Soc-Bitcoin*.

Ainsi, ces deux cas suggèrent que lorsque l'énumération des cliques dans les flots de liens ne termine pas (dans le temps imparti), c'est en raison d'un temps d'énumération des cliques dans les graphes G_t trop long. Par conséquent, pour améliorer l'énumération des cliques maximales dans les flots de liens et espérer la faire passer à l'échelle de ces jeux de données massifs que nous ne sommes pas en mesure de traiter, cela suggère qu'il serait d'abord nécessaire d'améliorer les algorithmes d'énumération des cliques dans les graphes, avant de les appliquer à un algorithme sur les flots de liens.

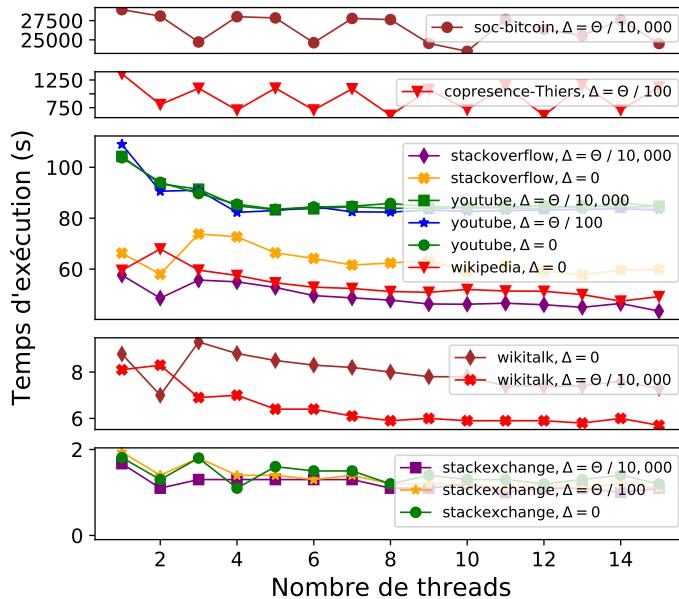
5.6.5 Expérimentations avec la version parallèle de l'algorithme

Pour terminer, nous étudions ici une version parallèle de notre implémentation, afin d'évaluer le gain en vitesse d'énumération qu'elle peut apporter. L'algorithme 5.1 est en effet facilement parallélisable, car les itérations de la boucle sur T à la ligne 1 sont indépendantes les unes des autres. Notre approche consiste à diviser l'intervalle de temps total de durée Θ du flot de liens en n_{th} sous-intervalles, où n_{th} est le nombre de *threads* sur lesquels nous effectuons la parallélisation. Pour chaque sous-intervalle en parallèle, le *thread* correspondant énumère toutes les cliques maximales qui commencent pendant cet intervalle, en suivant la boucle de la ligne 1. Nous choisissons de diviser l'intervalle de temps total du flot de liens de manière à ce qu'environ le même nombre de liens démarre dans chaque sous-intervalle. Ainsi, chaque *thread* traite environ $\frac{m}{n_{th}}$ liens. Or, selon l'expression établie par le théorème 5.5, la complexité de l'algorithme est en $\mathcal{O}(m \cdot 3^{d/3} \cdot 2^q \cdot d^2)$. Donc, en divisant le nombre de liens par n_{th} , la complexité de la version parallèle de l'algorithme devient en $\mathcal{O}\left(\frac{1}{n_{th}} \cdot m \cdot 3^{d/3} \cdot 2^q \cdot d^2\right)$ par *thread*.

La figure 5.10 illustre les résultats du processus de parallélisation sur les flots de liens massifs détaillés dans le tableau 5.2. Elle présente le temps d'exécution en fonction du nombre de *threads* utilisés. La figure 5.10 (a) montre les flots de liens pour lesquels la parallélisation offre une réduction intéressante des temps de calcul, tandis que la figure 5.10 (b) montre les flots de liens pour lesquels la parallélisation n'apporte aucune amélioration. Par exemple, la parallélisation de *Wikipedia* avec $\Delta = \Theta / 100$ conduit à une division par trois du temps d'énumération, et encore plus pour *Copresence-Thiers* avec $\Delta = 0s$, alors que le même processus sur les flots de liens de la figure 5.10 (b) n'apporte que très peu de gains. Nous observons également que pour presque tous les jeux de données, les temps de calcul ne diminuent pas de manière significative lorsque l'on utilise plus de 4 *threads*, contrairement à ce qu'on pourrait attendre.



(a) Flots de liens pour lesquels la parallélisation offre une réduction intéressante du temps de calcul.



(b) Flots de liens pour lesquels la parallélisation ne fonctionne pas.

FIGURE 5.10 – Temps de calcul (en secondes) de l’implémentation parallèle en C++, en fonction du nombre de *threads*, sur l’ensemble des flots de liens massifs détaillé dans le tableau 5.2. La durée Δ des liens est exprimée en fonction de Θ , la durée totale du flot de liens.

Cette faible accélération du temps de calcul peut s'expliquer par la distribution de la densité du flot de liens dans le temps. En effet, le temps de calcul est fixé par le *thread* sur lequel le calcul est le plus lent, et celui-ci correspond au sous-flot de liens le plus dense. C'est ce que l'on a vu par exemple dans le cas extrême de la figure 5.9, où le flot de liens considéré a des instants $t \in T$ qui génèrent beaucoup plus de cliques maximales que d'autres, jusqu'à en devenir bloquant. Dans ce cas, la parallélisation que nous proposons pourrait ne pas du tout fonctionner.

5.7 Conclusion

Dans ce chapitre, nous avons abordé le problème de l'énumération des cliques maximales dans les flots de liens. Notre contribution est multiple. Nous avons proposé un nouvel algorithme pour résoudre ce problème, qui passe à l'échelle de flots de liens massifs issus du monde réel ; nous avons analysé sa complexité, en fonction des caractéristiques de l'entrée et de la sortie de l'algorithme, en montrant notamment qu'elle est proche de la taille de la sortie ; nous avons réalisé des expériences approfondies sur différents jeux de données issus d'interactions réelles au cours du temps, afin de montrer que notre algorithme permet un gain de performance par rapport à l'état de l'art de plusieurs ordres de grandeur ; enfin, nous avons fourni une implémentation en Python et une en C++, dont nous avons discuté de sa parallélisation.

Les travaux effectués dans ce chapitre ouvrent plusieurs perspectives de recherche. La première piste consiste à approfondir l'étude des raisons pour lesquelles l'énumération prend plus de temps sur certains flots de liens, en particulier dans les cas où la limite de temps de notre protocole est atteinte. Par exemple, nous pouvons analyser le facteur $\frac{1}{r}$ qui apparaît dans la complexité et qui est lié à la sortie de l'algorithme. Cela nous permettrait de développer davantage les analyses que nous avons proposées à ce sujet afin de déterminer si le blocage provient de l'exploration d'un grand nombre de branches inutiles ou si le nombre de cliques maximales est tout simplement trop élevé. Une autre direction de recherche consisterait à améliorer le processus de parallélisation en affinant le partitionnement du flot de liens en fonction de propriétés structurelles à définir. Cela permettrait de mieux équilibrer le calcul entre les différents threads. Par exemple, l'étude de la structure du flot de liens pourrait nous aider à anticiper les instants $t \in T$ pour lesquels il y a plus ou moins de cliques maximales, et ainsi nous permettre de calculer des sous-intervalles plus adaptés et obtenir une meilleure accélération du calcul grâce à la parallélisation. Il serait également intéressant d'appliquer des méthodes de parallélisation pour l'énumération des cliques dans les graphes. En effet, nous avons observé que l'énumération des cliques peut être bloquante à l'intérieur des graphes instantanés G_t . Enfin, approfondir l'étude de l'effet de la durée des liens Δ sur l'énumération des cliques maximales pourrait permettre d'obtenir des indications pour l'analyse

des jeux de données en fonction des échelles de temps que l'on souhaite prendre en compte.

Plus largement, l'énumération d'autres types de sous-graphes denses dans les flots de liens est un problème pertinent. Comme on l'a vu, les sous-graphes denses permettent de définir des communautés, un concept largement utilisé pour décrire les structures de données d'interaction. À cette fin, il est intéressant d'énumérer l'ensemble des cliques de taille fixe k , car c'est un problème qui est différent de l'énumération des cliques maximales, et dont l'implémentation peut être plus rapide en pratique sur des graphes issus du monde réel, pour de petites valeurs de k . La méthode de percolation de k -cliques que nous avons étudiée dans le chapitre 3 a été étendue réseaux temporels [PBV07 ; Bou+18]. Nous proposons dans le chapitre 6 suivant de nous inspirer de la méthode utilisée dans ce chapitre, pour énumérer les k -cliques dans les flots de liens. Cela conduit à une définition originale de la percolation de clique temporelle, car le formalisme des flots de liens nous permet de généraliser facilement la notion d'adjacence de k -cliques définie dans les graphes. Nous pouvons alors tirer profit d'un algorithme rapide d'énumération de cliques pour obtenir efficacement une structure de communautés temporelles.

LSCPM : communautés temporelles par percolation de cliques dans les flots de liens

Résumé

Comme on l'a vu précédemment, la méthode de percolation de cliques (CPM), qui implique la percolation de k -cliques, est une technique qui offre plusieurs avantages pour la détection de communautés dans les graphes. Étant donné qu'il est important d'étudier les interactions qui se produisent au fil du temps dans divers contextes, la méthode de percolation de cliques dynamique (DCPM) a été proposée pour étendre la méthode CPM aux réseaux temporels. Cependant, les implémentations existantes ne permettent pas de traiter des jeux de données massifs.

Dans ce chapitre, nous proposons une nouvelle définition des communautés CPM dans les flots de liens, que nous appelons communautés LSCPM (*Clique Percolation Method in Link Streams*), accompagnée d'un algorithme pour les calculer. Nous commençons par proposer une extension des k -cliques pour le formalisme des flots de liens, et nous développons un algorithme pour les énumérer. Cela nous permet d'introduire une règle d'adjacence des cliques dans les flots de liens, pour définir les communautés par percolation de ces cliques. Les résultats de nos expérimentations sur des jeux de données issus du monde réels démontrent l'efficacité de notre méthode, capable de traiter des flots de liens massifs en un temps raisonnable.

Nous comparons cet algorithme à l'extension de la percolation de cliques qui a été proposée dans les graphes temporels vus comme des suites de graphes instantanés (DCPM). Notre nouvelle définition donne des communautés proches de celles de l'état de l'art et le temps de calcul pour les obtenir est plus rapide de plusieurs ordres de grandeur. Nous validons la pertinence de ces communautés dans des scénarios réels et mettons en évidence leur capacité à fournir des informations sur l'importance des nœuds dans les flots de liens.

6.1 Introduction

Nous nous intéressons ici à la méthode de détection de communautés par percolation de cliques (CPM), pour travailler à son extension aux réseaux temporels. Comme on l'a vu dans le chapitre 3, cette technique très étudiée présente des avantages significatifs (communautés qui se chevauchent, définition déterministe, communautés définies localement...). Or, malgré les implémentations existantes, son adaptation à un contexte dynamique ne permet pas de traiter des jeux de données massifs.

L'adaptation de la méthode CPM aux graphes temporels a en fait été proposée assez tôt [PBV07]. Nous appelons cette approche la *méthode de percolation de cliques dynamique* (DCPM pour *Dynamic Clique Percolation Method*). Cette approche consiste à mettre en œuvre l'algorithme CPM sur les graphes instantanés à chaque pas de temps. Cette méthode est intéressante du fait que la définition des communautés soit déterministe, car cela permet de suivre les communautés d'un pas de temps au suivant. Récemment, Boudebza *et al.* [Bou+18] ont introduit un algorithme plus rapide, appelé *Online Clique Percolation Method*, que nous appelons OCPM, qui améliore la mise à jour des communautés CPM à chaque pas de temps.

Comme la notion de clique a été étendue aux flots de liens [VLM16] et que nous avons développé une énumération efficace des cliques maximales (voir chapitre 5), il est désormais possible de s'en inspirer pour étudier les extensions de CPM aux flots de liens. Ce formalisme offre un rôle symétrique à la structure et au temps dans la représentation des données et cela en donne une vision singulière qui permet de définir directement l'adjacence des cliques temporelles. Nous pouvons ainsi introduire une nouvelle définition des communautés CPM temporelles, que nous appelons **LSCPM** (*Link Stream Clique Percolation Method*), et qui est distincte de celle des communautés DCPM. Cela permet de nous inspirer de la méthode efficace de calcul des communautés CPM dans les graphes, qui utilise la structure de l'Union-Find pour construire les communautés. L'algorithme obtenu est alors plus rapide que les implémentations DCPM de l'état de l'art, il est capable de traiter des données en ligne pour obtenir des communautés qui évoluent naturellement dans le temps. Nous verrons que les communautés LSCPM obtenues sont proches des communautés DCPM, plus agrégées dans le sens où une communauté LSCPM peut englober plusieurs communautés DCPM.

Les contributions de ce chapitre sont les suivantes : nous commençons par définir les k -cliques dans les flots de liens et définir leur adjacence pour former les communautés LSCPM (section 6.2). Puis, nous proposons un algorithme pour les énumérer (section 6.3). Ensuite, nous développons la structure de **Temporal Union-Find** une extension de la structure Union-Find pour manipuler des ensembles de sommets temporels de manière efficace (section 6.4). Tout cela nous permet de proposer notre algorithme de calcul de communautés LSCPM (section 6.5), et qui est directement

inspiré de l'algorithme CPM sur graphes, grâce au formalisme des flots de liens. Nous en donnons une implémentation en libre accès qui permet de traiter des flots de liens volumineux¹. Enfin, nous présentons dans la section 6.6 une étude expérimentale qui montre l'efficacité de notre algorithme LSCPM sur plusieurs cas réels, qui compare les communautés obtenues à celles de DCPM, et qui illustre sa pertinence pour tirer des informations sur les données réelles examinées.

6.2 Définitions et notations

6.2.1 Cliques dans les flots de liens

Nous rappelons qu'un flot de liens $L = (T, V, E)$ est défini par un intervalle de temps T , un ensemble de sommets V et un ensemble de liens temporels $E \subseteq T \times T \times V \times V$ (par exemple celui de la figure 6.1 (a)).

Nous utilisons la définition d'une clique dans un flot de liens qui reprend celle définie dans les préliminaires et utilisée dans le chapitre 5, avec une légère différence pour éviter que les cliques existent sur des intervalles de temps de longueur nulle : une **clique** est une paire $(C, [t_0, t_1])$, où $C \subseteq V$, $|C| \geq 2$ et $t_0, t_1 \in T$, $\mathbf{t}_0 < \mathbf{t}_1$, tel que pour tout $u, v \in C, u \neq v$, il existe un lien (b, e, u, v) dans E tel que $[t_0, t_1] \subseteq [b, e]$.

Une **k -clique** est une clique contenant k sommets. Remarquons que si $(C, [t_0, t_1])$ est une k -clique, alors $(C, [t'_0, t'_1])$ est aussi une k -clique pour tout t'_0, t'_1 tel que $t_0 \leq t'_0 < t'_1 \leq t_1$. Pour éviter les redondances d'énumération, nous nous intéressons aux k -cliques **maximales en temps**, i.e. dont l'intervalle d'existence $[t_0, t_1]$ est maximal. Nous les appelons k -cliques *maximales* :

Définition 6.1 (k -clique maximale). *Pour $k \in \llbracket 2, +\infty \rrbracket$, une **k -clique maximale** est une clique $(C, [t_0, t_1])$ contenant k sommets (i.e. $|C| = k$), et telle que son intervalle d'existence est maximal : il n'existe pas de $t'_0 < t_0$ ni de $t'_1 > t_1$ tels que $(C, [t'_0, t_1])$ ou $(C, [t_0, t'_1])$ est une clique du flot de liens.*

Avec cette définition, nous introduisons ci-dessous la notion d'adjacence de k -cliques, qui nous permet de généraliser les communautés CPM dans les graphes, aux flots de liens.

Définition 6.2 (k -cliques maximales adjacentes). *Deux k -cliques maximales $(C, [t_0, t_1])$ et $(C', [t'_0, t'_1])$ sont dites **adjacentes** lorsqu'elles partagent $k - 1$ sommets et se chevauchent sur un intervalle de temps de durée strictement positive, i.e. $|C \cap C'| = k - 1$ et $|(t_0, t_1) \cap (t'_0, t'_1)| > 0$, où $|I|$ représente la durée $b - a$ de l'intervalle $I = [a, b]$.*

La figure 6.1 (b) représente l'ensemble des k -cliques maximales du flot de liens de la figure 6.1 (a) pour $k = 3$. On y observe que certaines cliques sont adjacentes.

1. <https://gitlab.lip6.fr/baudin/lscpm>

C'est le cas par exemple de la clique $(\{d, e, f\}, [4, 9])$, qui est adjacente aux trois cliques $(\{c, d, e\}, [2, 13])$, $(\{e, f, g\}, [3, 5])$ et $(\{e, f, g\}, [8, 12])$, car elle partage avec elles une $(k - 1)$ -clique (*i.e.* un lien ici) sur une durée strictement positive.

6.2.2 Communautés dans les flots de liens

À présent, nous introduisons la notion de communauté temporelle dans le cas général des flots de liens. Dans un contexte dynamique, il est naturel de définir une **communauté temporelle** comme un ensemble de **sommets temporels** :

Définition 6.3 (Communauté temporelle). *Une communauté temporelle est un ensemble de sommets temporels de la forme (u, I) , où u est un sommet, et I est un ensemble d'intervalles de temps disjoints, qui sont les intervalles de temps pendant lesquels u est présent dans la communauté.*

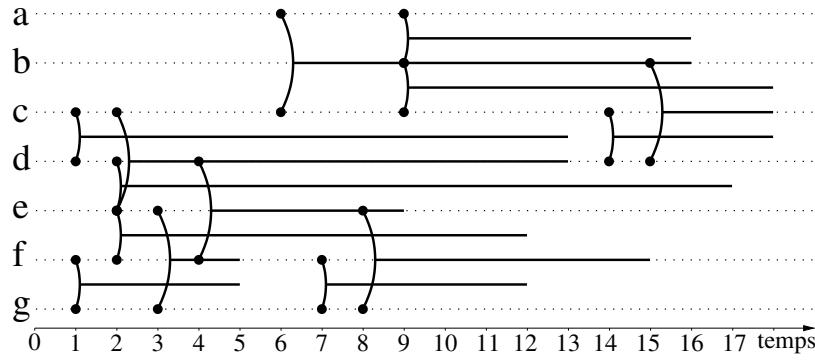
À partir de cette définition, la notion de communautés LSCPM est similaire à celle de communautés CPM dans les graphes, mais avec la notion d'adjacence de k -cliques adaptée aux flots de liens :

La figure 6.1 (c) représente l'ensemble des communautés LSCPM du flot de liens de la figure 6.1 (a) avec $k = 3$. Ce flot de liens contient deux communautés LSCPM, qui correspondent chacune à un regroupement de 3-cliques maximales adjacentes de la figure 6.1 (b).

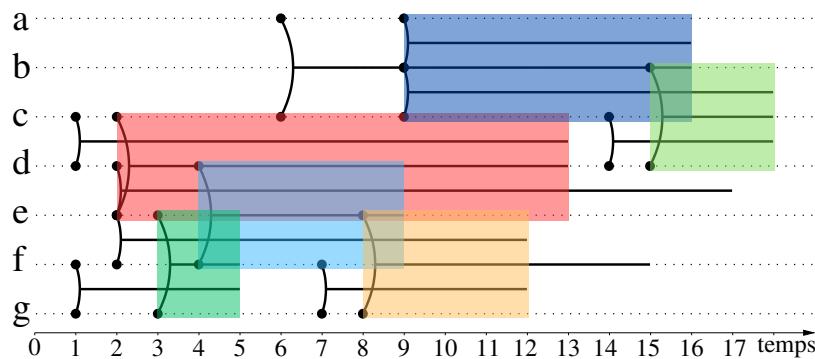
Définition 6.4 (Communauté LSCPM). *Une communauté LSCPM est une communauté temporelle dont les sommets temporels appartiennent à un ensemble maximal de k -cliques maximales, pouvant être reliées entre elles par une série de k -cliques maximales adjacentes.*

Cette définition appelle quelques observations. Premièrement, lorsque k augmente, les communautés ne peuvent que se diviser et/ou perdre des sommets temporels. En d'autres termes, si $k_1 < k_2$, alors chaque communauté LSCPM calculée avec k_2 est incluse dans une communauté LSCPM calculée avec k_1 . En effet, si deux k_2 -cliques c_1 et c_2 sont adjacentes, alors elles contiennent une $(k_2 - 1)$ -clique en commun. Commençons par remarquer que toutes les k_1 -clique de c_1 sont adjacentes, car c_1 est une clique, de même que toutes k_1 -clique de c_2 sont adjacentes. Or, c_1 et c_2 partagent des mêmes k_1 -cliques : celles qui se trouvent au sein de la $(k_2 - 1)$ -clique qu'elles ont en commun. Donc tous les sommets temporels de c_1 et c_2 appartiennent à la même communauté LSCPM définie avec k_1 . Cette propriété est illustrée et discutée dans la section 6.6.5.

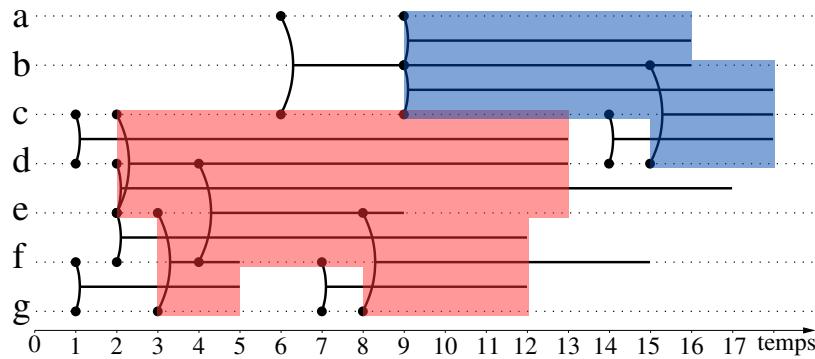
Deuxièmement, rappelons la définition des communautés de CPM dynamique (DCPM) introduite par Palla *et al.* [PBV07]. Ils considèrent le réseau temporel comme une suite de graphes instantanés mis à jour par la création et la suppression d'arêtes dans le temps. Un lien (b, e, u, v) appartient à tous les graphes instantanés de l'in-



(a) Exemple d'un flot de liens.



(b) Flot de liens de figure (a) et toutes ses 3-cliques maximales en couleur.



(c) Flot de liens de figure (a) et ses deux communautés LSCPM en couleur. La communauté en rouge est composée des sommets temporels $(c, [[2, 13]])$, $(d, [[2, 13]])$, $(e, [[2, 13]])$, $(f, [[3, 12]])$ et $(g, [[3, 5], [8, 12]])$. La communauté en bleu est composée des sommets temporels $(a, [[9, 16]])$, $(b, [[9, 16]])$, $(c, [[9, 18]])$ et $(d, [[15, 18]])$.

FIGURE 6.1 – Exemple d'un flot de liens avec ses k -cliques maximales pour $k = 3$ et les communautés LSCPM associées.

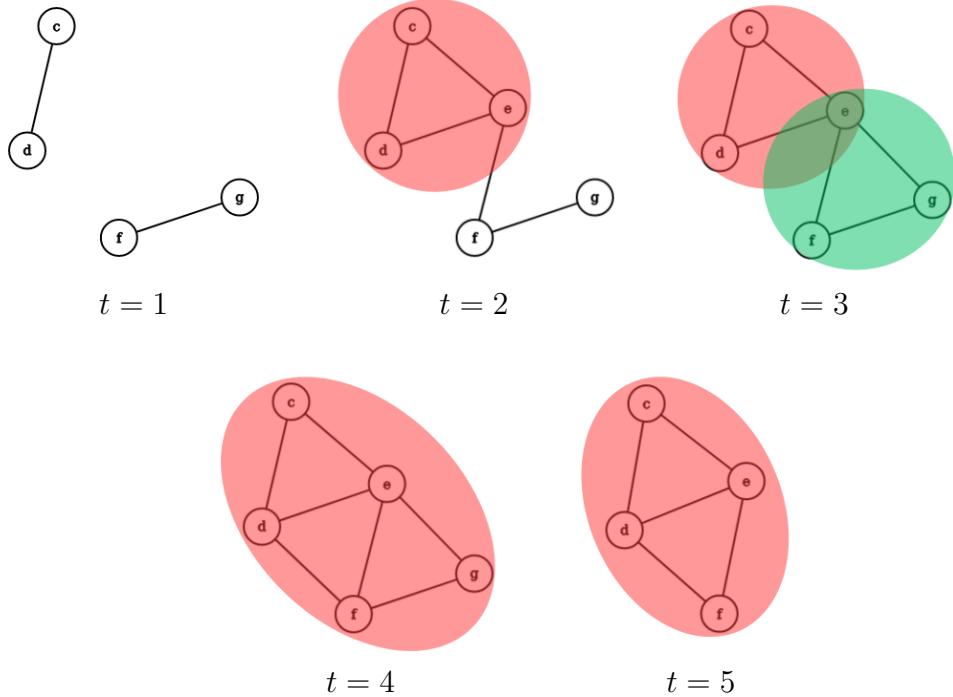
tervalle de temps $[b, e]$, et il est supprimé du graphe instantané au temps e . Le principe de la méthode DCPM consiste à calculer les communautés CPM pour chacun de graphes instantanés. Ensuite, la comparaison des communautés obtenues pour deux graphes instantanés consécutifs permet de déterminer dans quelle mesure chaque communauté évolue en gagnant des sommets, en en perdant, en mourant (en disparaissant ou en fusionnant avec une communauté plus importante) ou en naissant (en apparaissant ou en étant détachée d'une communauté plus importante). La figure 6.2 illustre ce processus. La figure 6.2 (a) représente les cinq premiers graphes instantanés du flot de liens de la figure 6.1, avec leurs communautés CPM en couleur. Une même couleur d'un graphe instantané à l'autre correspond à une même communauté, qui évolue dans le temps. Par exemple, de $t = 2$ à $t = 3$, la communauté rouge reste la même et la communauté verte naît. Au temps $t = 4$, ces deux communautés se fusionnent, faisant disparaître la communauté verte, et à $t = 5$ la communauté résultante perd un sommet.

Les communautés DCPM peuvent être considérées comme des communautés temporelles : étant donnés deux graphes instantanés consécutifs à t_i et t_{i+1} , on peut considérer qu'un sommet d'une communauté DCPM à t_i appartient en fait à la communauté sur l'intervalle de temps continu $[t_i, t_{i+1}]$, puisque dans leur définition les liens qui existent à t_i ne sont pas supprimés avant t_{i+1} . La figure 6.2 (b) représente les communautés DCPM du flot de liens de la figure 6.1. Sur l'intervalle de temps $[1, 6]$, elles correspondent aux communautés calculées de la figure 6.2 (a).

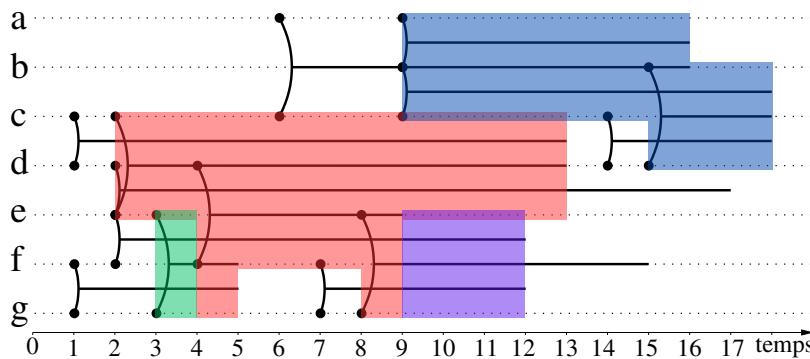
À partir de ce constat, il est important de noter qu'une communauté LSCPM est une union de communautés DCPM. En effet, les sommets d'une k -clique maximale ($C, [t_0, t_1]$) du flot de liens appartiennent à l'union des communautés DCPM de C dans chacun des graphes instantanés qui existent sur $[t_0, t_1]$ (à l'instant t_1 la clique n'existe plus dans le graphe instantané). Par exemple, la k -clique ($\{e, f, g\}, [3, 5]$) appartient à l'union des communautés de $\{e, f, g\}$ sur les graphes instantanés aux temps $t = 3$ et $t = 4$ de la figure 6.2 (a). Ainsi, les communautés DCPM rouge et verte de la figure 6.2 (b) sont incluses dans la communauté LSCPM rouge de la figure 6.1 (c). Il en est de même pour la communauté violette. Les communautés LSCPM sont donc des communautés DCPM agrégées. Nous étudions ce phénomène d'agrégation dans la section 6.6.3 de nos expériences.

6.3 Algorithme d'énumération des k -cliques dans les flots de liens

La première étape pour mettre en place notre algorithme de percolation de cliques dans les flots de liens est d'énumérer l'ensemble des k -cliques maximales. Pour cela, nous nous inspirons du travail présenté dans le chapitre 5 qui énumère les cliques maximales. L'idée principale est de ramener le calcul à l'énumération des k -cliques



(a) Graphes instantanés du flot de liens de la figure 6.1 (a), pour les 5 premiers temps où une arête commence ou se termine : $t = 1, 2, 3, 4, 5$. Les communautés DCPM, qui correspondent à l'évolution des communautés CPM d'un graphe instantané à l'autre, sont représentées en couleur.



(b) Flot de liens de figure 6.1 (a) et ses quatre communautés DCPM en couleur. Sur l'intervalle de temps $[1, 6[$, ce sont les communautés provenant de la figure (a).

FIGURE 6.2 – Communautés DCPM du flot de lien de la figure 6.1, avec $k = 3$.

du graphe à un instant t , car des algorithmes très efficaces ont été conçus pour cette tâche [DBS18 ; Li+20]. Ensuite, comme dans le chapitre 5, nous calculons les temps de début et de fin de chaque k -clique maximale du flot de liens induite par ces cliques de graphes. Le pseudocode de la procédure est donné par l'algorithme 6.1, et il est illustré par la figure 6.3 qui en donne un exemple d'exécution.

L'algorithme 6.1 énumère l'ensemble des k -cliques maximales du flot de liens d'entrée L . Pour chaque lien (b, e, u, v) , il considère uniquement le sous-flot des liens qui ont déjà été traités, et il énumère les k -cliques maximales qui sont créées en y ajoutant ce lien (b, e, u, v) . Pour ce faire, il établit un parallèle entre L et le graphe agrégé de ses liens G qui évolue au cours du temps. Il part d'un graphe vide $G = (V, \emptyset)$ (ligne 1), et par la boucle de la ligne 3, il traite les liens dans l'ordre chronologique d'apparition (ils sont ordonnés arbitrairement lorsqu'ils ont un même temps de début). Pour chaque lien (b, e, u, v) , il met à jour G , qui contient l'ensemble des arêtes des liens déjà traités et qui ne sont pas encore terminés au temps b (lignes 4 à 6). Ensuite, il énumère les k -cliques maximales créées par l'ajout de ce lien. Ce sont les cliques $(C, [t_0, t_1])$ telles que C est une k -clique de G qui contient u et v . Chacune de ces k -cliques maximales commence au temps b , car le lien entre u et v n'existe pas avant b , et dure aussi longtemps que tous ses liens existent, de sorte que son temps de fin est le minimum des temps de fin des arêtes qui la composent (ligne 9). Notez que chaque clique est bien énumérée, lorsque le dernier de ses liens dans l'ordre est traité, et qu'elle n'est bien énumérée qu'une seule fois, car elle n'existe pas encore lorsque ses autres liens, situés avant dans l'ordre, sont traités.

Algorithme 6.1: Énumérer les k -cliques maximales d'un flot de liens.

Entrée: Flot de liens $L = (T, V, E)$; $k \in \llbracket 3, +\infty \rrbracket$.

Sortie: Ensemble des k -cliques maximales de L .

```

1  $G \leftarrow (V, \emptyset)$  graphe vide
2  $\mathcal{E} \leftarrow$  tableau associatif vide           //  $\mathcal{E}$  : temps de fin des arêtes
3 for  $(b, e, u, v) \in E$  triés par  $b$  croissant do
4   Ajouter l'arête  $\{u, v\}$  à  $G$ 
5    $\mathcal{E}(\{u, v\}) \leftarrow e$                  // Enregistre le temps de fin de  $\{u, v\}$ 
6   Supprimer de  $G$  toutes les arêtes  $\{x, y\}$  telles que  $\mathcal{E}(\{x, y\}) < b$ 
7    $G_{Cliques} \leftarrow$  toutes les  $k$ -cliques de  $G$  qui contiennent  $u$  et  $v$ 
8   for  $C \in G_{Cliques}$  do
9      $end \leftarrow \min_{x, y \in C} (\mathcal{E}(\{x, y\}))$ 
10    output  $k$ -clique  $(C, [b, end])$ 
```

La complexité de l'algorithme 6.1 est donnée par le théorème 6.1.

Théorème 6.1 (Complexité de l'énumération des k -cliques maximales). *L'algorithme 6.1 énumère l'ensemble des k -cliques maximales du flot de liens d'entrée en*

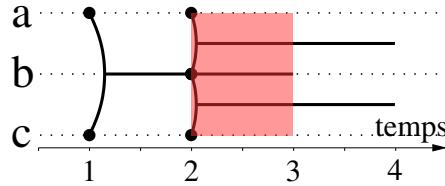


FIGURE 6.3 – Exemple d'une énumération d'une 3-clique maximale dans un flot de liens simple. Les liens sont traités dans l'ordre chronologique, par exemple $(1, 4, a, c)$, $(2, 4, a, b)$ et $(2, 4, b, c)$. Lorsque les deux premiers liens sont traités, aucune 3-clique n'est calculée. Lorsque le dernier lien $(2, 4, b, c)$ est traité, le graphe agrégé contient la 3-clique $\{a, b, c\}$. Comme elle contient l'arête $\{b, c\}$ correspondant à ce lien, elle est renvoyée. Son temps de début est alors 2, et son temps de fin est le minimum du temps de fin de ses arêtes, c'est-à-dire 3. La 3-clique renvoyée par l'algorithme est donc $(\{a, b, c\}, [2, 3])$, et elle n'est bien énumérée qu'une seule fois.

temps $\mathcal{O}\left(m \cdot k^3 \cdot \left(\frac{d}{2}\right)^{k-2} + m \cdot d^2 + m \cdot \log(m)\right)$, où m est le nombre de liens et d le degré instantané maximal d'un nœud, c'est-à-dire son nombre maximal de voisins à un instant donné.

Démonstration. Considérons une itération de la boucle commençant à la ligne 3, et (b, e, u, v) le lien associé.

Tout d'abord, montrons que la mise à jour de G par les lignes 4 à 6 est effectuée en $\mathcal{O}(\log(m))$. G est stocké sous forme de listes d'adjacences, associant à chaque sommet l'ensemble de ses voisins, de sorte que l'ajout d'une arête à la ligne 4 est effectué en temps constant. La ligne 5 s'exécute également en temps constant. Enfin, trouver toutes les arêtes dont le temps de fin est inférieur à b (ligne 6) se fait en maintenant une liste des liens triée par leur temps de fin dans G , ce qui est en $\mathcal{O}(\log(m))$ pour chaque nouveau lien.

Ensuite, intéressons-nous à la ligne 7, qui énumère l'ensemble des cliques de G qui contiennent u et v . Cela revient à énumérer les $(k-2)$ -cliques du sous-graphe induit par les voisins communs de u et v , que l'on note $G(N(u) \cap N(v))$. Ce sous-graphe induit se calcule en $\mathcal{O}(d^2)$. Puis, l'énumération des $(k-2)$ -cliques de $G(N(u) \cap N(v))$ dépend de la valeur de k . Si $k = 3$, cela consiste à énumérer les sommets, ce qui se fait en $\mathcal{O}(d)$. Si $k = 4$, il s'agit d'énumérer les arêtes, ce qui est en $\mathcal{O}(d^2)$. Si $k \geq 4$, nous utilisons l'algorithme d'énumération de k -cliques dans les graphes présenté par Danisch *et al.* [DBS18]. Dans cet article, le théorème 5.7 donne la complexité de l'énumération en $\mathcal{O}\left(k \cdot m \cdot \left(\frac{d}{2}\right)^{k-2}\right)$ dans un graphe avec m liens et un degré maximal de d . Ainsi, l'énumération des $(k-2)$ -cliques de $G(N(u) \cap N(v))$ se fait en $\mathcal{O}\left((k-2) \cdot d^2 \cdot \left(\frac{d}{2}\right)^{k-4}\right)$ et la complexité globale de la ligne 7 est en $\mathcal{O}\left(k \cdot \left(\frac{d}{2}\right)^{k-2}\right)$, quelle que soit la valeur de k . Notons que cette valeur fixe une borne supérieure au

nombre de k -cliques énumérées par l’itération de la boucle.

Chacune des k -cliques de G calculées à la ligne 7 est ensuite traitée par la boucle de la ligne 8, en $\mathcal{O}(k \cdot (k - 1)) = \mathcal{O}(k^2)$. La complexité totale de ces opérations est donc en $\mathcal{O}\left(k^3 \cdot \left(\frac{d}{2}\right)^{k-2}\right)$.

Nous obtenons finalement le résultat en additionnant les trois complexités ci-dessus, et en les multipliant par m pour tenir compte de chacun des m tours de boucles de la ligne 3. \square

Notez que ce résultat peut être légèrement affiné en remplaçant k^3 par $k \cdot (k - 1) \cdot (k - 2)$. En effet, dans la preuve, pour simplifier la formule de complexité, nous avons remplacé un facteur $k - 2$ et un facteur $k - 1$ par un facteur k . Cela peut avoir un impact significatif, étant donné que les valeurs de k que nous utilisons sont généralement petites (typiquement inférieures à 10). En pratique, c’est la valeur de k qui détermine l’efficacité de l’énumération, car les facteurs k^3 et $\left(\frac{d}{2}\right)^{k-2}$ montrent que cette méthode ne reste efficace que pour de petites valeurs de k .

6.4 Temporal Union-Find : adapter l’Union-Find à des ensembles d’éléments temporels

L’algorithme LSCPM que nous allons présenter est une adaptation aux flots de liens de l’algorithme CPM, vu dans la section 3.2. Nous avons vu que l’algorithme CPM reposait en grande partie sur la structure d’Union-Find, permettant de faire des unions d’ensembles disjoints, et de savoir à quel ensemble appartient un élément de manière très efficace. Il s’en sert pour identifier les communautés CPM à des ensembles disjoints de $(k - 1)$ -cliques. Dans le cadre des flots de liens, nous souhaitons adapter cette structure de telle manière qu’elle puisse traiter des ensembles d’éléments temporels, c’est-à-dire qu’un élément existe dans un ensemble pendant un ou plusieurs intervalles de temps donnés. Nous verrons dans la suite que cela nous permet d’identifier les communautés LSCPM à des ensembles de $(k - 1)$ -cliques temporelles, disjointes dans le temps.

Nous introduisons donc ce que l’on appelle la structure de **Temporal Union-Find**, qui représente des ensembles d’éléments temporels. Un élément x peut appartenir à des ensembles différents, mais sur des intervalles de temps disjoints ; il peut également appartenir à un même ensemble pendant plusieurs intervalles de temps disjoints. La Temporal Union-Find est une forêt d’arbres, où chaque arbre représente un ensemble. Si un élément x est dans un ensemble pendant un intervalle de temps $[t_0, t_1]$, alors x est associé à un nœud q de l’arbre de cet ensemble, pendant cet intervalle de temps.

La structure de Temporal Union-Find associe à chaque sommet x l'ensemble $\text{TUF.id}(x)$ des couples $(q, [t_0, t_1])$ tels que q est un nœud de l'Union-Find, et x appartient à l'ensemble représenté par l'arbre contenant q pendant l'intervalle de temps $[t_0, t_1]$. Tous les intervalles de temps sont disjoints, de telle sorte que x n'appartienne pas à plusieurs ensembles à un même instant. Un exemple de Temporal Union-Find représentant deux ensembles d'éléments temporels est donné par la figure 6.4.

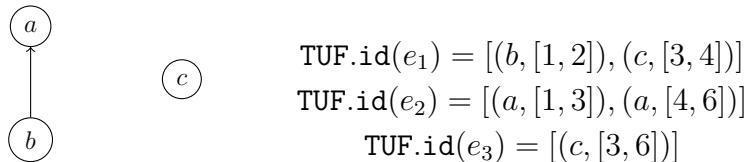


FIGURE 6.4 – Exemple d'une instance de Temporal Union-Find qui représente les deux ensembles $\{(e_1, [1, 2]), (e_2, [1, 3]), (e_2, [4, 6])\}$ et $\{(e_1, [3, 4]), (e_3, [3, 6])\}$. On note que e_1 appartient à ces deux ensembles, sur des intervalles de temps différents, et que e_2 appartient au premier ensemble pendant deux intervalles de temps disjoints.

Opérations

Une instance de Temporal Union-Find TUF étend les opérations de l'Union-Find de la manière suivante :

- $\text{TUF.Union}(p, q)$: réalise l'union des arbres dont les racines sont p et q , en connectant p à q ou q à p , de la même manière que dans la structure d'Union-Find.
- $\text{TUF.Find}(x, t)$ renvoie l'identifiant de l'ensemble auquel x appartient au temps t . L'appel renvoie $(q, [t_0, t_1])$, où q est la racine d'un arbre de la Temporal Union-Find, et $[t_0, t_1]$ est l'intervalle de temps pendant lequel x appartient à l'ensemble identifié par q , avec $t \in [t_0, t_1]$. On suppose que si cette fonction est appelée, alors x appartient bien à un ensemble au temps t .
- $\text{TUF.Add}(x, [t_0, t_1], p)$: ajoute x à l'ensemble identifié par p , pendant l'intervalle de temps $[t_0, t_1]$. Si x est déjà dans l'ensemble identifié par p , pendant un intervalle de temps $[t'_0, t'_1]$ tel que $[t_0, t_1] \cap [t'_0, t'_1] \neq \emptyset$, alors après cette opération, il y appartient pendant $[t_0, t_1] \cup [t'_0, t'_1]$.
- $\text{TUF.MakeSet}()$: crée un nouvel ensemble vide, en créant un nouvel arbre contenant un nœud q , et renvoie q .

Complexité des opérations faites dans l'ordre chronologique

Considérons une instance de Temporal Union-Find TUF . Pour calculer les complexités des opérations décrites ci-dessus, nous faisons l'hypothèse que les opérations TUF.Add sont réalisées dans l'ordre chronologique. Ce sera le cas de notre algorithme

LSCPM. Alors, si x est un élément donné, les entrées $(q, [t_0, t_1])$ de $\text{TUF.id}(x)$ sont triées par ordre de t_0 croissant. Ainsi, on peut tester en $\mathcal{O}(1)$ si x est bien dans un ensemble de TUF à un certain instant t donné ($x \in \text{TUF à } t$) : grâce à l'ordre chronologique, t est supérieur ou égal au temps de début de chaque intervalle dans $\text{TUF.id}(x)$, et il suffit alors de regarder le **dernier** élément de $\text{TUF.id}(x)$, qui est de la forme $(q, [t_0, t_1])$, et de tester si $t \in [t_0, t_1]$ ou non (et si $t \notin [t_0, t_1]$, c'est nécessairement que $t > t_1$). Une fois que l'on sait que $x \in \text{TUF à } t$, on peut réaliser l'opération $\text{TUF.Find}(x, t)$, qui consiste alors à faire un Find d'Union-Find classique sur le nœud q , en $\mathcal{O}(1)$ (voir le chapitre 3 où nous avons introduit la structure Union-Find, et où nous avons vu que cette opération pouvait être en temps constant avec une bonne implémentation de la structure). De même que pour l'Union-Find classique, les opérations d'Union et de MakeSet se font en $\mathcal{O}(1)$. Enfin, l'opération $\text{TUF.Add}(x, q, [t_0, t_1])$ est différente selon que x est déjà dans l'ensemble de q sur un sous-intervalle commun à $[t_0, t_1]$ ou non. Si c'est le cas, alors du fait de l'ordre chronologique, c'est le dernier élément de $\text{TUF.id}(x)$ qui est en commun : il vaut $(q', [t'_0, t'_1])$ où q' est dans le même arbre que q et $t'_0 \leq t_0 \leq t'_1$. Alors, on remplace l'entrée $(q', [t'_0, t'_1])$ par $(q, [t'_0, \max(t_1, t'_1)])$, ce qui se fait en $\mathcal{O}(1)$. Sinon, c'est nécessairement que le dernier élément $(q', [t'_0, t'_1])$ de $\text{TUF.id}(x)$ (s'il existe) est terminé à t_0 , *i.e.* il est tel que $t'_1 < t_0$. Il suffit alors d'ajouter l'entrée $(q, [t_0, t_1])$ à la fin de $\text{TUF.id}(x)$, ce qui se fait également en $\mathcal{O}(1)$.

Ainsi, toutes les opérations de la Temporal Union-Find sont en $\mathcal{O}(1)$ lorsque les données sont traitées par ordre chronologique. Comme dans le cas de CPM, cela garantit d'avoir une méthode efficace pour construire les communautés à la volée à partir de l'énumération des k -cliques. Les communautés temporelles peuvent ainsi être représentées comme des ensembles de $(k - 1)$ -cliques temporelles : les éléments de TUF sont des ensembles de $k - 1$ sommets qui appartiennent à des communautés pendant un ou plusieurs intervalles de temps disjoints.

6.5 LSCPM : algorithme CPM adapté aux flots de liens

À partir de notre algorithme d'énumération des k -cliques maximales dans les flots de liens, et de la structure de Temporal Union-Find, nous pouvons à présent utiliser l'algorithme 3.1 CPM pour l'étendre aux flots de liens : les communautés LSCPM sont construites à la volée, en étant stockées comme l'ensemble des $(k - 1)$ -cliques des k -cliques maximales qui les composent. Une $(k - 1)$ -clique de la k -clique $(C_k, [t_0, t_1])$ est de la forme $(C_{k-1}, [t_0, t_1])$, avec $C_{k-1} \subseteq C_k$ contenant $k - 1$ sommets. Nous appelons cette procédure LSCPM, et elle est donnée par l'algorithme 6.2. Nous le décrivons dans la section 6.5.1, et nous en donnons la complexité dans la section 6.5.2.

6.5.1 Pseudocode

L'algorithme 6.2 traite chaque k -clique maximale une par une, en suivant l'ordre chronologique de leur temps de début. Cet ordre est donné par l'énumération de l'Algorithme 6.1. Pour chaque k -clique maximale $(C_k, [t_0, t_1])$, les ensembles de sommets C_{k-1} de ses $(k-1)$ -cliques $(C_{k-1}, [t_0, t_1])$ sont traités un par un (ligne 5). Pour cela, l'algorithme vérifie si C_{k-1} appartient à une communauté (lignes 6 et 7), et si ce n'est pas le cas, il en crée une nouvelle (ligne 9). Avant de faire la fusion à la ligne 11, il s'assure par la ligne 10 que C_{k-1} appartient bien à la communauté pendant tout l'intervalle de temps de la k -clique $[t_0, t_1]$. Notez que, comme pour l'algorithme CPM, la première Union entre -1 et q renvoie directement q .

Algorithme 6.2: LSCPM : percolation de cliques dans les flots de liens.

Entrée: Flot de liens $L = (T, V, E)$; $k \in \llbracket 3, +\infty \rrbracket$.
Sortie: Temporal Union-Find représentant les communautés LSCPM de L .

```

1 TUF ← structure Temporal Union-Find vide
2 for each  $k$ -clique maximale  $(C_k, [t_0, t_1])$  triées par  $t_0$  croissant do
3    $p \leftarrow -1$ 
4   for each  $u \in C_k$  do
5      $C_{k-1} \leftarrow C_k \setminus \{u\}$ 
6     if  $C_{k-1} \in \text{TUF}$  à  $t_0$  then
7        $q \leftarrow \text{TUF.Find}(C_{k-1}, t_0)$ 
8     else
9        $q \leftarrow \text{TUF.MakeSet}()$ 
10      TUF.Add( $C_{k-1}, [t_0, t_1], q$ )
11       $p \leftarrow \text{TUF.Union}(p, q)$ 
```

La figure 6.5 donne un exemple de mise à jour de la structure de Temporal Union-Find TUF lors de l'exécution de l'algorithme 6.2 sur le flot de lien de la figure 6.1. Dans les figures 6.5 (a) et 6.5 (b), trois $(k-1)$ -cliques sont ajoutées à la structure TUF. Dans la figure 6.5 (c), les communautés de i_1 et i_4 sont fusionnées, car la k -clique contient une $(k-1)$ -clique dans l'arbre de i_1 et une autre dans l'arbre de i_4 ; $\{d, f\}$ est ajoutée, $\{e, f\}$ est étendue dans le temps, et $\{d, e\}$ reste inchangée car elle est déjà présente dans la communauté sur un intervalle de temps plus long (mais le $\text{TUF.Find}(\{d, e\})$ modifie son noeud i_3 en i_1). Dans la 6.5 (d) $\{e, f\}$ est étendue dans le temps, et comme les deux autres, $\{e, g\}$ et $\{f, g\}$, n'étaient plus dans la communauté, elles sont rajoutées sur l'intervalle de temps de la k -clique, $[8, 12]$.

Notons que la sortie de l'algorithme 6.2 est une structure de Temporal Union-Find, et qu'un post-traitement est nécessaire pour produire les communautés en elles-mêmes, sous formes d'ensembles de sommets temporels. Ce post-traitement se fait en prenant chaque ensemble C_{k-1} de TUF.id, et en parcourant TUF.id(C_{k-1}) qui

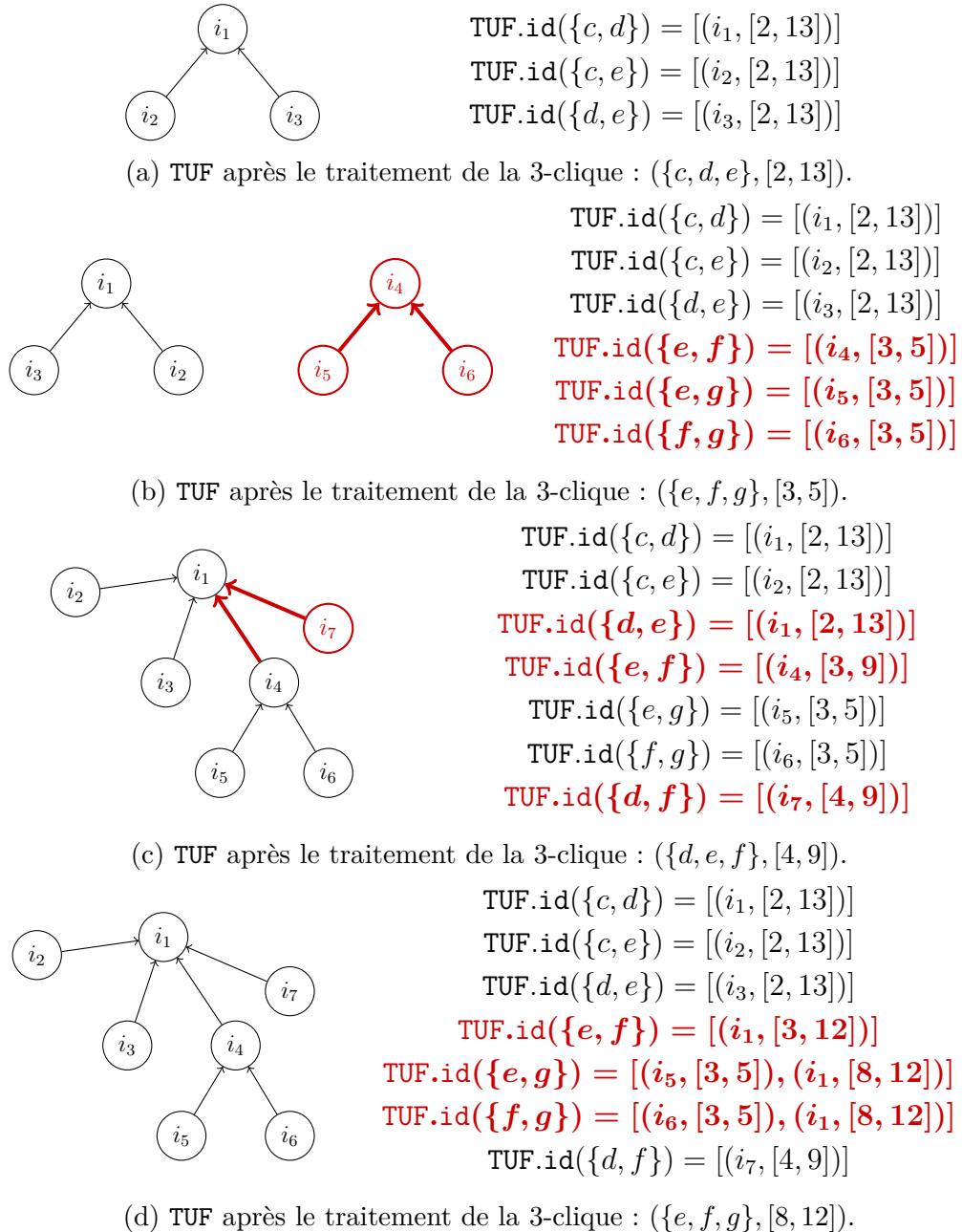


FIGURE 6.5 – Exemple de mises à jour de la Temporal Union-Find TUF de l’algorithme 6.2, pendant le traitement des quatre 3-cliques maximales de la communauté LSCPM rouge du flot de liens de la figure 6.1 : $(\{c, d, e\}, [2, 13])$, $(\{e, f, g\}, [3, 5])$, $(\{d, e, f\}, [3, 9])$ et $(\{e, f, g\}, [8, 12])$.

donne les noeuds de TUF qui lui sont associés avec leur intervalle de temps $(q, [t_0, t_1])$. Il suffit alors pour chacun de ces $(q, [t_0, t_1])$ de faire un appel à $\text{TUF.Find}(q)$ afin de connaître l'identifiant de sa communauté, puis d'ajouter chaque sommet de C_{k-1} dans cette communauté pendant l'intervalle de temps $[t_0, t_1]$.

6.5.2 Complexité

La complexité de l’Algorithme 6.2 est donnée par le théorème 6.2 suivant.

Théorème 6.2 (Complexité de l’algorithme LSCPM). *La complexité en temps de l’algorithme 6.2 est en $\mathcal{O}(k^2 \cdot n_k + c(k))$, où n_k est le nombre de k -cliques du flot de liens, et $c(k)$ la complexité de l’énumération des k -cliques, donnée par le théorème 6.1. Elle est donc en $\mathcal{O}\left(m \cdot k^3 \cdot \left(\frac{d}{2}\right)^{k-2} + m \cdot d^2 + m \cdot \log(m)\right)$, où d est le degré maximal d’un graphe instantané du flot de liens et m le nombre de liens.*

Démonstration. Considérons une k -clique maximale $(C_k, [t_0, t_1])$ correspondant à une itération de la boucle de la ligne 2. La ligne 4 effectue une itération par sommet de C_k , soit k itérations. Or, on a vu dans la section 6.4 où nous avons défini la structure de Temporal Union-Find que toutes les opérations de TUF se font en temps constant. En revanche, elles nécessitent d'accéder à la valeur associée à C_{k-1} dans le tableau associatif TUF.id . Comme C_{k-1} contient $k - 1$ sommets, cette opération se fait en $\mathcal{O}(k - 1)$. Ainsi, la boucle de la ligne 4 s'exécute en $\mathcal{O}(k - 1)$, et donc l'algorithme 6.2 s'exécute en $\mathcal{O}(k \cdot (k - 1) \cdot n_k) \subseteq \mathcal{O}(k^2 \cdot n_k)$.

En plus de cela, nous devons prendre en compte la complexité de l’énumération des k -cliques du flot de liens, $c(k)$. Nous obtenons ainsi une complexité globale de l’algorithme 6.2 en $\mathcal{O}(k^2 \cdot n_k + c(k))$.

Enfin, nous avons vu dans la preuve du théorème 6.1 qu'à chaque itération de la boucle de la ligne 3 de l’algorithme 6.1, le nombre de k -cliques énumérées est en $\mathcal{O}\left(k \cdot \left(\frac{d}{2}\right)^{k-2}\right)$. Comme il y a m itérations, on obtient que le nombre de k -cliques n_k est en $\mathcal{O}\left(m \cdot k \cdot \left(\frac{d}{2}\right)^{k-2}\right)$, d'où la deuxième partie du théorème, en combinant la borne ci-dessus sur n_k et la complexité de l’énumération des k -cliques maximales donnée par le théorème 6.1.

Notez que la complexité n'est pas changée par le post-traitement que nous avons décrit, qui construit les communautés temporelles à partir de la TUF. En effet, TUF contient au plus k entrées $(q, [t_0, t_1])$ par k -clique maximale de L (car la ligne 9 est appelée au plus k fois par k -clique maximale), il y en a donc au plus $k \cdot n_k$. Chacune de ces entrées $(q, [t_0, t_1])$ est dans une liste $\text{TUF.id}(C_{k-1})$, et on a vu qu'on y réalisait un appel $\text{TUF.Find}(q)$, qui donne une communauté temporelle à laquelle ajouter les $k - 1$ sommets de l’ensemble C_{k-1} pendant l’intervalle de temps $[t_0, t_1]$; cela se fait en $\mathcal{O}(k - 1)$. Le post-traitement est donc lui aussi en $\mathcal{O}(k^2 \cdot n_k)$.

□

Nous voyons grâce à ce théorème que notre algorithme est efficace dans la manière de traiter chaque k -clique, une fois qu'elles ont été calculées. En effet, chaque k -clique contient k $(k-1)$ -cliques, et il n'est donc pas possible de les traiter en moins de $k \cdot n_k$ opérations, en utilisant une approche similaire à la nôtre. De plus, on remarque que la complexité donnée par la deuxième partie du théorème est la même que celle de l'algorithme 6.1 d'énumération des k -cliques maximales donnée par le théorème 6.1. Cela indique que la complexité théorique des opérations décrites par l'algorithme 6.2 LSCPM ne dépassent pas celle de l'énumération des k -cliques maximales. Ainsi, tant qu'il n'existe pas d'algorithme d'énumération des k -cliques maximales n'ayant pas une meilleure complexité, calculer les communautés LSCPM à partir du flot des k -cliques maximales est mis en œuvre de manière optimale par l'algorithme 6.2, dans le sens où on ne peut pas obtenir une complexité théorique plus faible avec cette approche.

La seconde partie du théorème montre que la complexité dépend notamment d'un facteur cubique en k et du facteur $\left(\frac{d}{2}\right)^{k-2}$. Cela implique que cette complexité est intéressante surtout pour de petites valeurs de k . Néanmoins, nous verrons dans la section 6.6 que de petites valeurs de k sont suffisantes pour observer des propriétés intéressantes sur nos jeux de données, tout en permettant une construction rapide des communautés.

Par ailleurs, en pratique, l'algorithme 6.2 nécessite de stocker en mémoire tous les noeuds de l'Union-Find, et il peut y en avoir de l'ordre du nombre de $(k-1)$ -cliques du flot de liens. Cela peut être contraignant, par exemple si le jeu de données d'entrée contient une très grande clique. En effet, comme on l'a vu dans le chapitre 3, s'il existe une clique contenant 1 000 sommets, et que nous recherchons des communautés de 6-cliques, alors il y a plus de 10^{15} 5-cliques à stocker à partir de cette grande clique. Néanmoins, nous savons que les données issues d'interactions réelles ne présentent généralement pas beaucoup de grandes cliques, ce qui rend l'approche par k -cliques pertinente pour leur étude. Les besoins en mémoire ne se sont pas révélés prohibitifs au cours de nos expériences.

6.6 Étude expérimentale

Pour l'étude expérimentale, nous avons implémenté notre algorithme en Python et le code est disponible publiquement². Tout au long de cette section, nous fixons $k = 3$ sauf indication contraire. Nous verrons que cette valeur permet un calcul rapide tout en étant suffisante pour fournir des informations intéressantes sur les

2. <https://gitlab.lip6.fr/baudin/lscpm>

jeux de données. Nous présentons également dans la section 6.6.5 l'impact de l'augmentation de k sur la structure des communautés ; cela induit des communautés plus petites et permet donc de cibler leur cœur, avec plus ou moins de robustesse en fonction de la valeur de k .

Tout au long de cette section, une communauté temporelle est représentée dans les figures par ses sommets en ordonnée et le temps en abscisse. Un sommet qui appartient à la communauté sur un intervalle de temps donné est représenté par une barre colorée située au niveau de son ordonnée et qui recouvre les abscisses de l'intervalle de temps. Nous en donnons un exemple dans la figure 6.6.

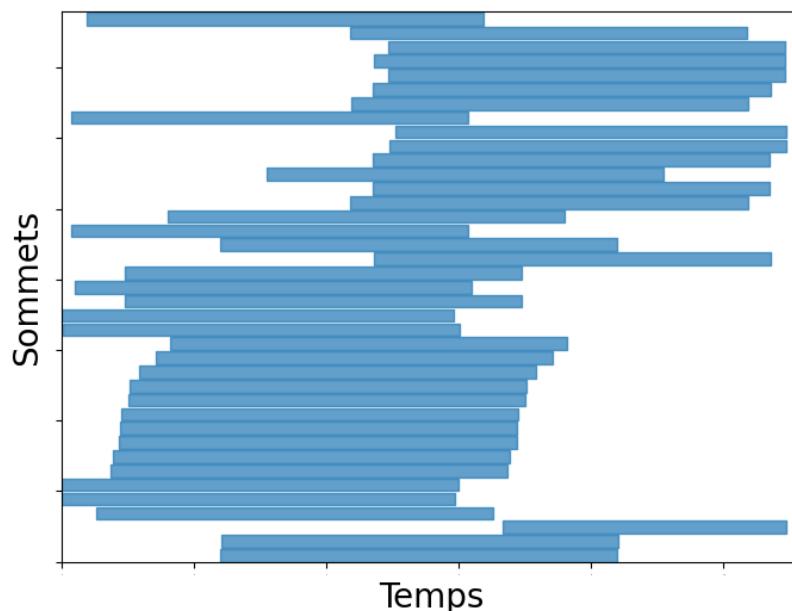


FIGURE 6.6 – Exemple d'une communauté temporelle. Les sommets sont en ordonnée, et le temps en abscisse. Une barre colorée correspond à un intervalle de temps pendant lequel le sommet associé appartient à la communauté.

6.6.1 Jeux de données

Nous avons effectué nos expériences sur des flots de liens issus du monde réel, de différentes tailles et impliquant différents types d'interactions. Bien que de nombreux jeux de données soient constitués intrinsèquement de liens avec une durée, dans de beaucoup cas, ces données sont enregistrées à des intervalles de temps discrets réguliers, en raison du protocole pratique d'acquisition des données. C'est le cas, par exemple, des données relatives aux contacts entre individus, généralement recueillies à l'aide capteurs Bluetooth. Par conséquent, la majorité des données est disponible sous forme de liens instantanés, avec des liens de la forme (t, u, v) , où u et v sont des sommets qui interagissent à l'instant t . Nous transformons alors ces flots de

liens en ajoutant une durée Δ créant des liens de la forme $(t, t + \Delta, u, v)$, comme fait précédemment pour l'énumération des cliques maximales dans les flots de liens, dans le chapitre 5. On notera que la valeur de Δ a un impact sur le nombre et la durée des cliques dans le flot de liens. En pratique, nous choisissons des valeurs uniformes de Δ qui sont cohérentes avec les échelles de temps typiques des interactions des jeux de données étudiés. Ces valeurs, bien que cohérentes, restent arbitraires et nous les utilisons pour démontrer l'efficacité et la pertinence de notre algorithme. Les utilisateurs peuvent ajuster ces valeurs en fonction et de la nature des jeux de données étudiés.

Les jeux de données sur lesquels nous avons réalisé nos expériences sont décrits dans le tableau 6.1. *Households* est un flot de liens qui représente les contacts entre les membres de cinq ménages dans un village du Kenya en 2012 [Kit+16] ; *Highschool* correspond aux contacts entre les élèves de cinq classes préparatoires à Marseille (PC, PC*, PSI*, MP, MP*) pendant une semaine en 2012 [FB14b] et *Infectious* est constitué de contacts entre les visiteurs d'un musée à Dublin en 2009 [Ise+11]. Ces trois jeux de données représentent des contacts entre individus, pour lesquels nous avons choisi de prendre une durée de liens de $\Delta = 1$ heure. Le jeu de données *Foursquare* est extrait de l'application éponyme, où les utilisateurs s'enregistrent dans les lieux qu'ils visitent, situés à New-York dans notre cas [Yan+14]. Il peut donc être représenté comme un flot de liens biparti entre les visiteurs et les lieux, daté par les heures des enregistrements. Dans ce flot de liens, nous fixons $\Delta = 6$ heures, puis nous le projetons sur l'ensemble des lieux : si un utilisateur est connecté à deux lieux au cours d'intervalles de temps qui se chevauchent, cela crée un lien entre ces lieux pendant la durée du chevauchement. Si les intervalles de temps de deux liens créés de cette manière se superposent, ils sont fusionnés en un seul lien. Ainsi, deux lieux sont connectés sur l'intervalle $[t_0, t_1]$ s'il existe un lien entre au moins un même visiteur et ces deux lieux à tout moment de cet intervalle de temps. Enfin, le flot de liens *Wikipedia* représente les liens entre les pages de Wikipedia, datés par l'heure de création du lien, sur plusieurs années dans les années 2000 [Mis09]. Il est difficile de fixer un Δ pertinent sur ce jeu de données, nous choisissons une durée de $\Delta = 1$ semaine, essentiellement pour explorer comment notre méthode s'adapte à des flots de liens massifs.

Nous avons vu dans le théorème 6.2 que la complexité de l'algorithme LSCPM était directement reliée au nombre de k -clique, avec le facteur $k \cdot n_k$. Dans le tableau 6.2, nous indiquons le nombre de k -cliques pour chaque jeu de données, pour k allant de 3 à 7. Cela permet d'anticiper les différences de temps de calcul entre les jeux de données, qui sont détaillées dans la section 6.6.2. Nous remarquons que pour les grands jeux de données, n_k augmente avec k , certainement parce que ces jeux de données contiennent quelques grandes cliques.

Flot de liens	Δ	m	n	d	D	r
<i>Households</i>	1 heure	2 136	75	19	3 jours	1h
<i>Highschool</i>	1 heure	5 528	180	18	8 jours	20s
<i>Infectious</i>	1 heure	44 658	10 972	43	3 mois	5 min
<i>Foursquare</i>	6 heures	268 472	33 153	81	10 mois	15 min
<i>Wikipedia</i>	1 semaine	38 953 380	1 870 709	33 217	2,3 ans	20s

TABLEAU 6.1 – Flots de liens de notre jeu de données. Δ est la durée des liens, m le nombre de liens, n le nombre de noeuds, d le degré instantané maximal, D la durée totale entre le premier et le dernier lien et r la résolution temporelle, c'est-à-dire la plus petite durée entre le début de deux liens.

Flot de liens	$k = 3$	$k = 4$	$k = 5$	$k = 6$	$k = 7$
<i>Households</i>	3 951	4 721	3 929	2 324	987
<i>Highschool</i>	2 468	583	97	11	1
<i>Infectious</i>	79 836	128 157	202 181	274 181	300 850
<i>Foursquare</i>	571 768	2 423 011	17 823 050	155 466 085	1 302 290 726
<i>Wikipedia</i>	3 757 877	1 148 832	1 763 386	4 545 105	11 853 134

TABLEAU 6.2 – Nombre de k -cliques (n_k) pour chaque jeu de données et pour k variant de 3 à 7.

6.6.2 Performances de LSCPM et passage à l'échelle de données massives

Nous comparons ici notre algorithme à celui de DCPM en termes de temps d'exécution³. Le temps d'exécution de DCPM est obtenu avec la meilleure implémentation disponible [Bou+18]. Nous appelons cette implémentation OCPM (pour Online CPM). Notez que cette comparaison se fait à titre indicatif, et que les méthodes DCPM et LSCPM ne fournissent pas exactement le même ensemble de communautés. Nous comparons les communautés sorties dans la section 6.6.3 suivante.

Dans notre implémentation de LSCPM, les k -cliques sont envoyées par l'algorithme d'énumération sous forme de flux vers l'entrée standard de l'algorithme LSCPM, qui les lit au fur et à mesure de l'énumération. Ces deux opérations sont effectuées sur deux *threads* différents ; mais à des fins de comparaison avec le temps d'exécution de DCPM, nous mesurons le temps de calcul global comme étant la somme des temps de calcul sur ces deux *threads*.

Nous avons réalisé toutes les expériences sur une machine Linux, équipée de deux processeurs Intel Xeon Silver 4210R avec vingt coeurs chacun, à 2,40 Ghz, et avec

3. Notre comparaison se concentre sur le temps d'exécution et non sur la complexité théorique, car les complexités de la méthode DCPM ou de l'implémentation OCPM ne sont pas données par leurs auteurs.

252 Go de RAM.

Le tableau 6.3 présente les temps de calcul des communautés avec l'implémentation OCPM de DCPM et notre implémentation LSCPM, sur tous les jeux de données du tableau 6.1, pour k allant de 3 à 7. Ces valeurs sont également représentées dans la Figure 6.7 à des fins de lisibilité. Nous observons que LSCPM est significativement plus rapide, en particulier sur les jeux de données de grande taille. Par exemple, avec $k = 3$ ou $k = 4$, il faut quelques secondes avec LSCPM pour calculer les communautés de *Foursquare*, alors qu'il faut plusieurs heures avec OCPM. De plus, pour le flot de liens massif *Wikipedia*, OCPM n'est pas en mesure de calculer l'ensemble des communautés en une semaine, alors que notre algorithme fournit les communautés en moins de 30 minutes pour toutes les valeurs de k testées. Notre algorithme permet donc d'étudier une structure communautaire dans des jeux de données massifs pour lesquels l'état de l'art ne fournit pas de résultat.

Flot de liens	$k = 3$		$k = 4$	
	OCPM	LSCPM	OCPM	LSCPM
<i>Households</i>	1,5s	0,1s	1,0s	0,1s
<i>Highschool</i>	3,6s	0,1s	1,9s	0,1s
<i>Infectious</i>	10min49s	1,4s	6min12s	3,3s
<i>Foursquare</i>	3h01min	9,2s	2h28min	43s
<i>Wikipedia</i>	-	13min44s	-	15min29s

Flot de liens	$k = 5$		$k = 6$		$k = 7$	
	OCPM	LSCPM	OCPM	LSCPM	OCPM	LSCPM
<i>Households</i>	0,7s	0,2s	0,6s	0,2s	0,5s	0,2s
<i>Highschool</i>	1,6s	0,1s	1,3s	0,1s	1,3s	0,1s
<i>Infectious</i>	3min58s	6,2s	3min02s	17,2s	2min30s	16,2s
<i>Foursquare</i>	2h12min	6min39s	2h08min	1h15mins	2h07min	12h35min
<i>Wikipedia</i>	-	15min44s	-	17min38s	-	23min39s

TABLEAU 6.3 – Temps de calcul des communautés OCPM et LSCPM en secondes, pour tous nos jeux de données, avec k variant de 3 à 7. Le symbole “-” signifie que le temps de calcul dépasse une semaine.

Un autre point intéressant est que le temps de calcul de LSCPM augmente avec k , alors qu'il diminue avec OCPM. Cela s'explique par le fait que la méthode OCPM obtient ses résultats en agrégeant les cliques maximales de taille *au moins* k , alors que notre méthode énumère des k -cliques. Plus k est grand, moins il y a de cliques maximales à énumérer et à traiter, d'où la diminution du temps de calcul. En revanche, nous avons vu dans la section 6.6.1 que n_k augmente généralement avec k pour les grands jeux de données, ce qui implique que le temps de calcul de LSCPM augmente également, conformément au théorème 6.2. Remarquez cependant que malgré cela, il n'y a qu'un seul cas où OCPM est plus rapide que LSCPM : le flot

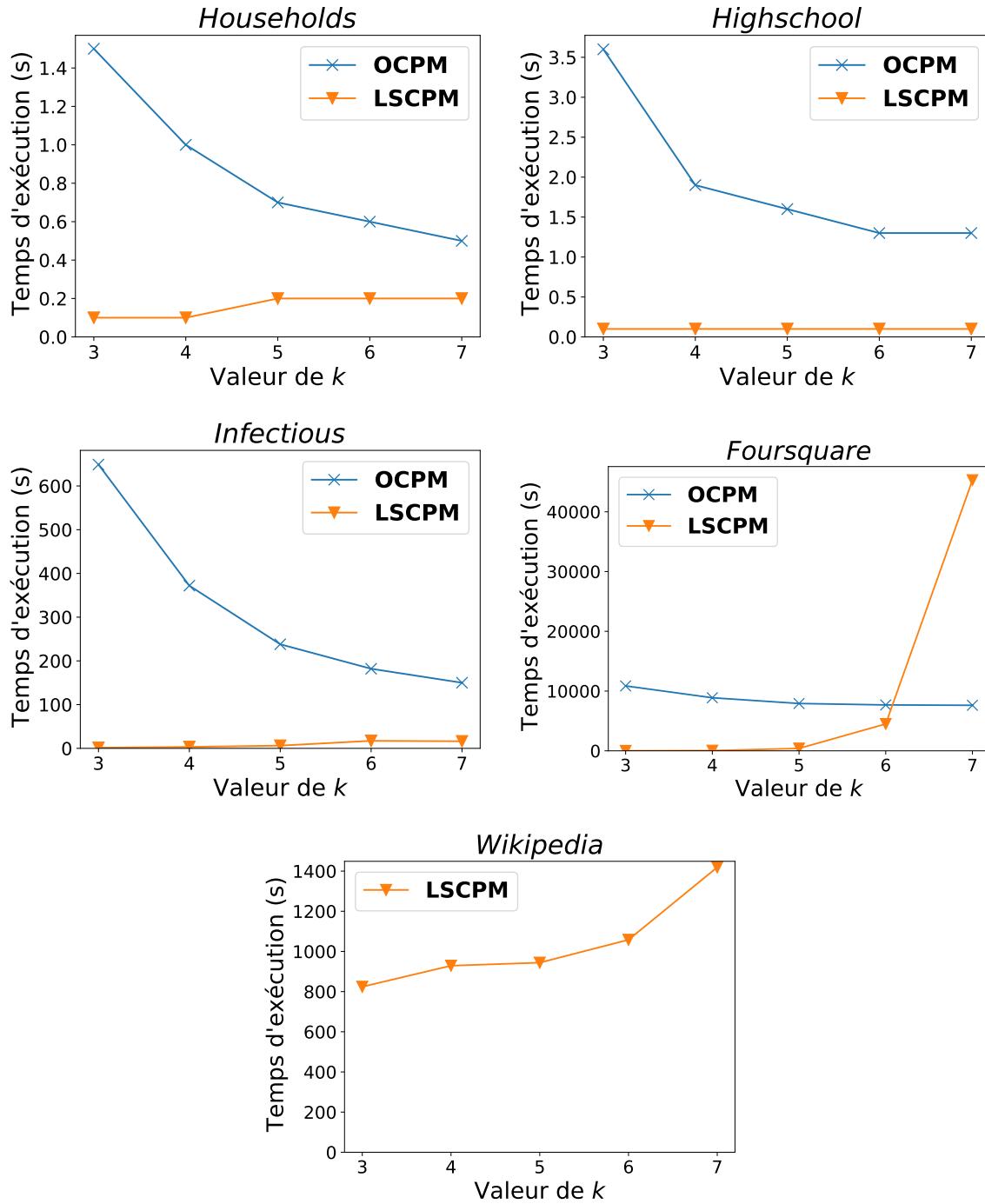


FIGURE 6.7 – Durées d'exécution des implémentations OCPM et de LSCPM pour chaque jeu de données. Les valeurs sont celles du tableau 6.3.

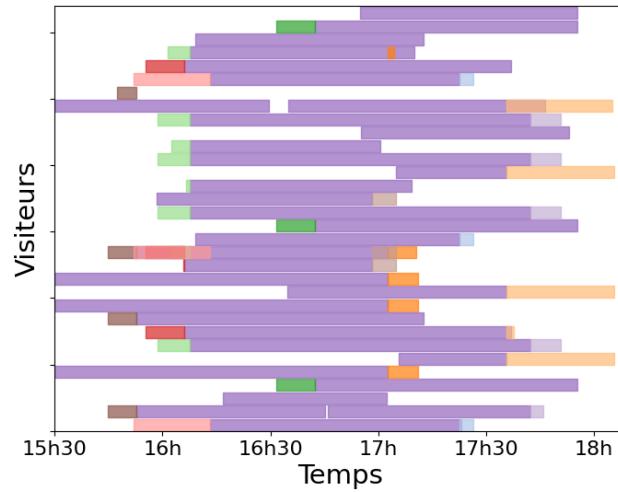
de liens de *Foursquare* avec $k = 7$, qui, comme nous l'avons vu, contient un très grand nombre de k -cliques.

6.6.3 Comparaison entre les communautés LSCPM et DCPM

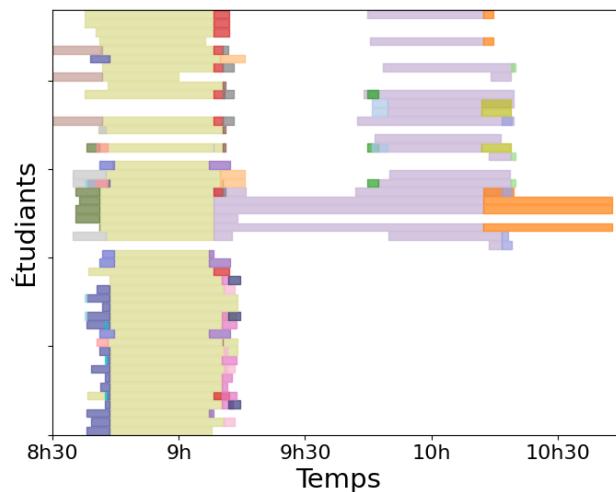
Dans ce qui suit, nous comparons les communautés obtenues avec notre algorithme LSCPM à celles obtenues avec DCPM, sur les quatre jeux de données où l'implémentation OCPM fournit un résultat. Notons qu'à notre connaissance, il n'existe pas de méthode de référence pour comparer les communautés temporelles qui se chevauchent. Nous n'utilisons donc pas d'outils tels que la *Normalized Mutual Information* ou l'indice de Rand qui sont utilisés pour comparer des partitions de sommets dans un graphe et qui nécessiteraient donc une adaptation au contexte considéré dans ce chapitre.

Nous avons vu dans la section 6.2 que chaque communauté DCPM est contenue dans une communauté LSCPM et, inversement, que chaque communauté LSCPM peut être considérée comme l'union de communautés DCPM. Cette propriété est illustrée par la Figure 6.8, qui donne deux exemples de communautés LSCPM. Chacune des communautés DCPM contenues dans la communauté LSCPM est représentée dans une couleur différente. La figure 6.8 (a) représente une communauté LSCPM d'*Infectious*. On observe qu'elle est composée de la communauté DCPM violette qui contient une grosse partie des sommets temporels, et de plusieurs autres petites communautés DCPM qui lui sont agrégées. La figure 6.8 (b) montre une communauté LSCPM de *Hightschool*. On voit qu'il y a deux communautés DCPM de taille importante qui sont agrégées, ainsi que plusieurs petites communautés. Il est intéressant de constater qu'il y a une signification temporelle à ces agrégations, car on voit que le premier bloc est lié au deuxième, par trois étudiants qui sont restés en contact pendant toute la période considérée. Dans la suite, nous étudions plus en détails dans quelle mesure les communautés DCPM sont regroupées en communautés LSCPM.

Pour évaluer la similarité entre une communauté LSCPM et les communautés DCPM qu'elle contient, nous calculons la taille relative (en nombre de sommets) de la plus grande communauté DCPM que chaque communauté LSCPM contient. La figure 6.9 (a) présente la distribution cumulative de cette valeur. Nous voyons clairement un pic à la dernière valeur, qui montre que pour tous les jeux de données, 90% des communautés LSCPM contiennent autant de sommets que leur plus grande communauté DCPM. Seules 1% des communautés LSCPM ont moins de 70% de leurs sommets dans leur plus grande communauté DCPM et aucune n'en a moins de 50%. Ces observations sont similaires pour les valeurs de k plus élevées que nous avons testées. Dans les deux exemples de la Figure 6.8, on distingue nettement une communauté DCPM plus grande que les autres, qui contient presque tous les sommets de la communauté LSCPM associée.



(a) Une communauté LSCPM de *Infectious*, et les communautés DCPM qu'elle contient (une couleur pour chacune).



(b) Une communauté LSCPM de *Highschool*, et les communautés DCPM qu'elle contient (une couleur pour chacune).

FIGURE 6.8 – Deux exemples de communautés LSCPM avec $k = 3$, où sont représentées en couleur les communautés DCPM qu'elles contiennent.

Nous observons également que chaque communauté LSCPM ne contient qu'un petit nombre de communautés DCPM. C'est ce qu'illustre la Figure 6.9 (b) qui représente la distribution cumulative du nombre de communautés DCPM que chaque communauté LSCPM contient. Dans tous les cas, plus de 70% des communautés LSCPM ne contiennent que 1 ou 2 communautés DCPM, et presque jamais plus de 10. Il y a cependant quelques exceptions : dans *Highschool*, 2,6% des communautés contiennent entre 10 et 26 communautés DCPM, et dans *Infectious*, 1,8% des communautés LSCPM contiennent entre 50 et 115 communautés DCPM. Par exemple, les communautés LSCPM de la figure 6.8 contiennent toutes les deux plus de 10 communautés DCPM.

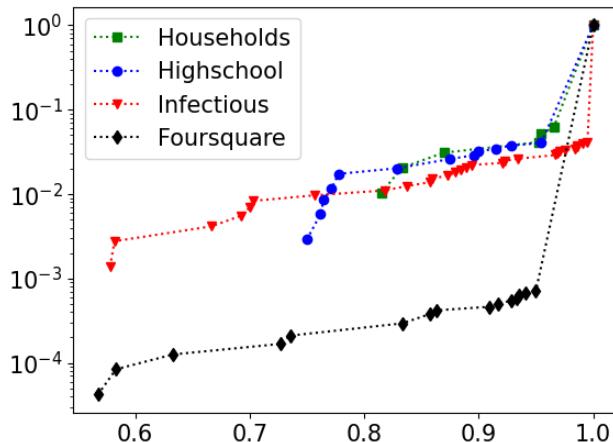
Par ailleurs, nous observons qu'il y a moins de petites communautés LSCPM que de petites communautés DCPM : nous constatons que *Households* a 17% plus de communautés DCPM de taille inférieure ou égales à 5 que de communautés LSCPM de tailles inférieures ou égales à 5, *Highschool* en a deux fois plus et *Infectious* en a six fois plus ; cependant les ensembles de communautés LSCPM et DCPM sont très similaires pour *Foursquare*. Cela indique que les petites communautés DCPM ont tendance à être regroupées dans des communautés LSCPM plus importantes, comme on peut le voir dans les exemples de la figure 6.8. Ces observations donnent un schéma typique du lien entre les communautés LSCPM et les communautés DCPM : une communauté LSCPM est en général composée d'une grande communauté DCPM qui contient presque tous les sommets, et éventuellement de quelques petites communautés résiduelles. L'interprétation que nous en faisons est que la plupart des informations véhiculées par les communautés obtenues dans les deux cas sont étroitement liées, mais que la méthode LSCPM permet de faciliter l'analyse des communautés en regroupant les petites communautés, moins significatives en termes d'interprétation temporelle, dans des communautés plus grandes.

6.6.4 Informations fournies par les communautés LSCPM

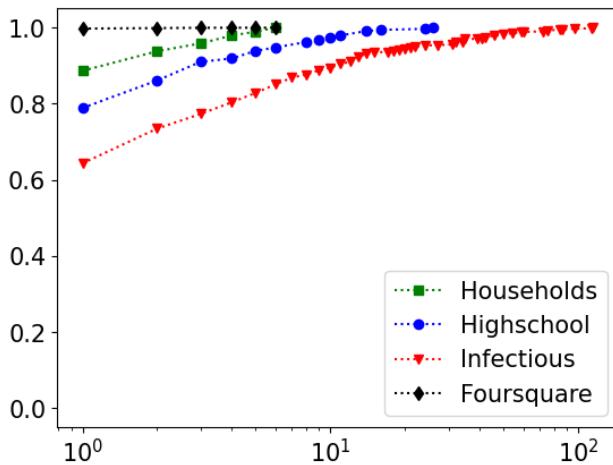
Pour étudier l'information au sein des communautés temporelles obtenues, nous disposons de métadonnées associées à nos jeux de données : les familles de *Households*, les classes de *Highschool*, et les coordonnées GPS des lieux de *Foursquare*.

Dans le cas des jeux de données *Households* et *Highschool*, qui sont basés sur des contacts entre personnes, nous observons que les communautés sont homogènes au niveau de ces catégories, comme on pouvait s'y attendre. En effet, dans le cas du jeu de données *Households*, 95% des communautés sont composées de membres d'une seule famille et les 5% restants de membres de deux familles. Dans le cas de *Highschool*, 70% des communautés sont composées d'une seule classe, 23% de deux classes, 6% de trois classes et 1% de quatre classes.

Ces métadonnées fournissent également des informations pertinentes sur les interactions entre individus au fil du temps, en indiquant qui fréquente qui et à quel



(a) Distribution cumulative de la taille relative de la plus grande communauté DCPM dans chaque communauté LSCPM.



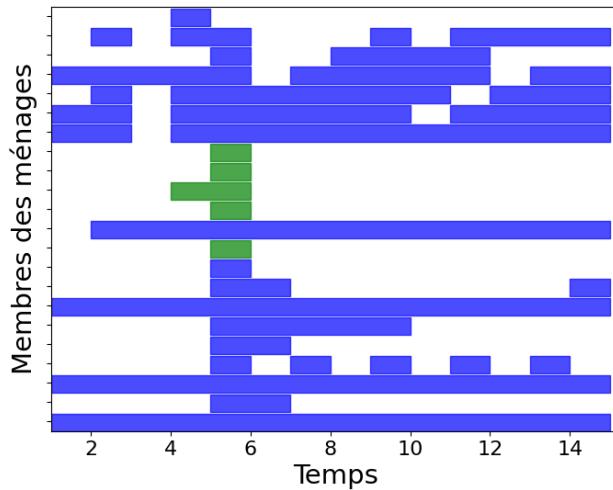
(b) Distribution cumulative du nombre de communautés DCPM par communauté LSCPM (échelle logarithmique en abscisse).

FIGURE 6.9 – Composition des communautés LSCPM en termes de communautés DCPM, avec $k = 3$.

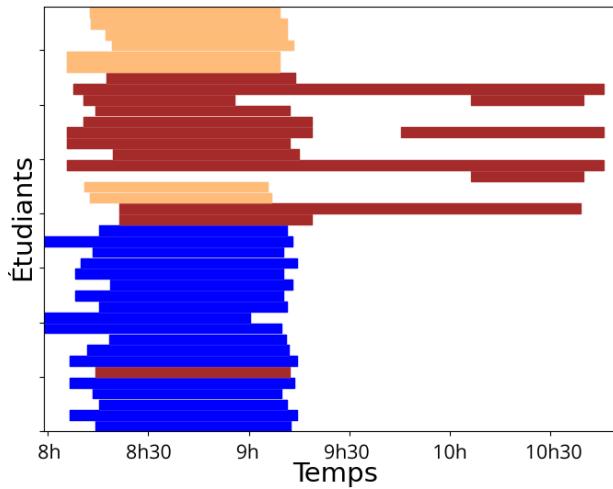
moment. Par exemple, la figure 6.10 (a) représente une communauté de *Households*, composée de 5 membres d'une famille (en vert) et de 17 membres d'une autre famille (en bleu). Elle met en évidence l'existence d'une rencontre d'au plus une heure entre toutes ces personnes, sauf une. De même, la figure 6.10 (b) montre une communauté dans laquelle nous observons des membres de trois classes différentes de *Highschool*, qui sont les trois classes de physique de l'école préparatoire (PC, PC*, PSI*). On y distingue deux périodes : pendant la première, les élèves des trois classes sont regroupés, ce qui suggère une période où les élèves peuvent se rencontrer et se mélangler. Ensuite, la communauté se réduit à quelques noeuds de la classe marron, ce qui peut indiquer la proximité de ces étudiants pendant les cours ou les groupes de travail.

En ce qui concerne *Foursquare*, nous pouvons utiliser les métadonnées pour étudier la distribution géographique des lieux visités par les mêmes personnes. La figure 6.11 montre une carte d'une partie de la ville de New-York affichant un échantillon de quatre communautés LSCPM. Nous observons que certaines d'entre elles sont relativement regroupées géographiquement, comme la communauté verte autour de Times Square ou la communauté rouge qui semble correspondre au quartier financier dans le sud de Manhattan. D'autres sont plus étendues, comme la communauté bleue dans la banlieue ouest de New-York. Cela se produit lorsque plusieurs personnes se déplacent d'une partie de la ville à l'autre au cours d'une période Δ . Ainsi, ici, Δ permet d'ajuster l'extension géographique des communautés, puisque des Δ plus faibles devraient correspondre à des extensions géographiques plus petites.

Il peut également être pertinent d'évaluer l'implication des sommets par rapport aux communautés auxquelles ils appartiennent. En effet, certains appartiennent à plus de communautés que d'autres, ce qui permet d'identifier des noeuds particulièrement importants au sein d'un groupe ou qui jouent un rôle de passerelles entre les groupes. La figure 6.12 illustre ces deux aspects : il s'agit de la distribution cumulative du nombre de communautés auxquelles appartient chaque sommet. Les points situés tout à gauche correspondent aux sommets qui n'appartiennent à aucune communauté. Sur ce point, les différents jeux de données donnent des résultats très différents. Par exemple, dans *Foursquare*, environ 20% des sommets n'appartiennent à aucune communauté ; cela correspond à des lieux où les utilisateurs visitent moins d'autres lieux en commun au cours de la période considérée. Nous observons ce phénomène pour certains lieux spécifiques tels que les centres médicaux, les bureaux, les terrains de jeux... En revanche, dans *Households*, chaque sommet appartient à au moins deux communautés, ce qui est raisonnable puisqu'il s'agit d'un réseau de contacts entre les membres d'une même famille, qui interagissent donc beaucoup entre eux. Dans *Highschool*, nous constatons que la plupart des noeuds appartiennent à de nombreuses communautés, ce qui est également logique puisque les élèves sont regroupés en classes et que chaque jour crée de nouvelles communautés. Enfin, les



(a) Une communauté de *Households* dont les sommets sont colorés en fonction de la famille d'appartenance.



(b) Une communauté de *Hightschool* dont les sommets sont colorés en fonction de la classe.

FIGURE 6.10 – Exemples de communautés LSCPM. Les couleurs proviennent des métadonnées.

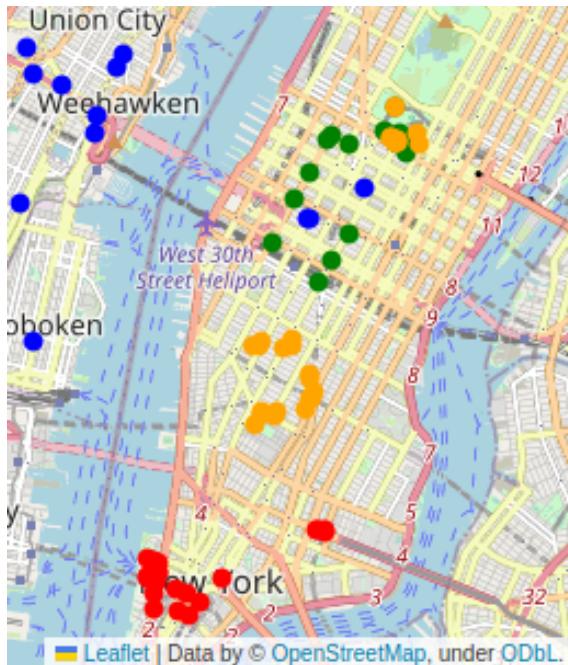


FIGURE 6.11 – Une carte de Manhattan (NYC), où quatre communautés *Foursquare* sont représentées par des couleurs différentes.

sommets qui appartiennent à de nombreuses communautés se trouvent tout à droite de la distribution. C'est particulièrement flagrant dans le cas de *Foursquare*, où près de 10% des sommets appartiennent à plus de 10 communautés, et quelques-uns à plus de 100. Ces derniers peuvent être décrits comme des nœuds centraux du flot de liens, qui interagissent avec de nombreux autres nœuds tout au long de la période d'observation. Par exemple, le lieu de *Foursquare* qui appartient de loin au plus grand nombre de communautés (1,5 fois plus que le deuxième) est la célèbre *Pennsylvania station*, qui est la plus importante gare ferroviaire intercités de la ville de New York.

6.6.5 Influence de k sur la structure de communautés

La taille k des cliques qui constituent les communautés LSCPM est le paramètre clé de l'algorithme. Nous discutons ici de l'effet de l'augmentation de k sur la structure des communautés, afin de donner une intuition à l'utilisateur quant au choix à faire pour cette valeur.

Comme nous l'avons vu dans la section 6.2, si $k_1 < k_2$, alors chaque communauté calculée avec k_2 est incluse dans une communauté calculée avec k_1 . Cela signifie qu'en augmentant k , les communautés se divisent et/ou perdent des sommets temporels. La figure 6.13 donne un exemple de ce phénomène sur une communauté du jeu de données *Foursquare* : nous voyons la communauté calculée avec $k = 3$ et comment

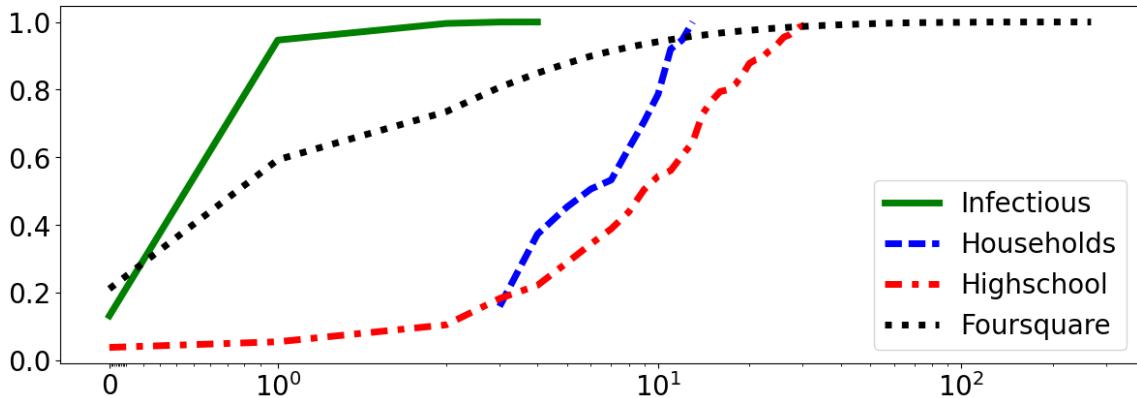


FIGURE 6.12 – Distribution cumulative du nombre de communautés auxquelles chaque sommet appartient. Notez que l’axe des abscisses est en échelle logarithmique, sauf entre 0 et 1, afin de montrer les sommets qui n’appartiennent à aucune communauté.

elle se divise et perd des noeuds lorsque l’on augmente k jusqu’à 7. Nous observons que la communauté reste presque identique pour $k = 4$, qu’elle se divise en trois communautés pour $k = 5$, et que l’une de ces petites communautés se divise à nouveau pour $k = 7$. Nous observons également que certains noeuds qui appartiennent à la communauté pour $k = 3$ à un instant donné n’appartiennent à aucune des communautés résultantes à cet instant pour des valeurs plus grandes de k . Ainsi, l’augmentation de k conduit à des communautés plus densément connectées, mais plus petites en taille et plus courtes dans le temps. Cela permet de modifier la granularité des communautés dynamiques qui peuvent être obtenues sur un jeu de données en se concentrant sur le “cœur” des interactions.

Grâce à cela, nous pouvons identifier des sous-communautés pertinentes lorsque des métadonnées sont disponibles. Par exemple, les sommets de la communauté *Foursquare* de la figure 6.13 correspondent à 6 types de lieux (sur les 261 disponibles), tous liés au sport : *Athletic-&-Sport*, *Bike-Shop*, *Stadium*, *Sporting-Goods-Shop*, *Gym-/Fitness-Center*, et *Motorcycle-Shop* (la plupart des autres communautés présentent des étiquettes plus variées). Nous cherchons à savoir si la division des communautés lorsque k augmente se traduit par une sélection des types de lieux. Pour $k = 5$, la communauté verte possède ces 6 étiquettes, mais les noeuds des communautés bleue et rouge n’ont que 3 étiquettes : *Sporting-Goods-Shop*, *Gym-/Fitness-Center* et *Bike-Shop*. De plus, pour $k = 7$, la communauté verte se divise en deux parties, dont l’une se concentre sur les sports à deux roues : *Motorcycle*, *Bike*, *Stadium*, ce qui montre que cette décomposition permet de déterminer les intérêts communs des utilisateurs.

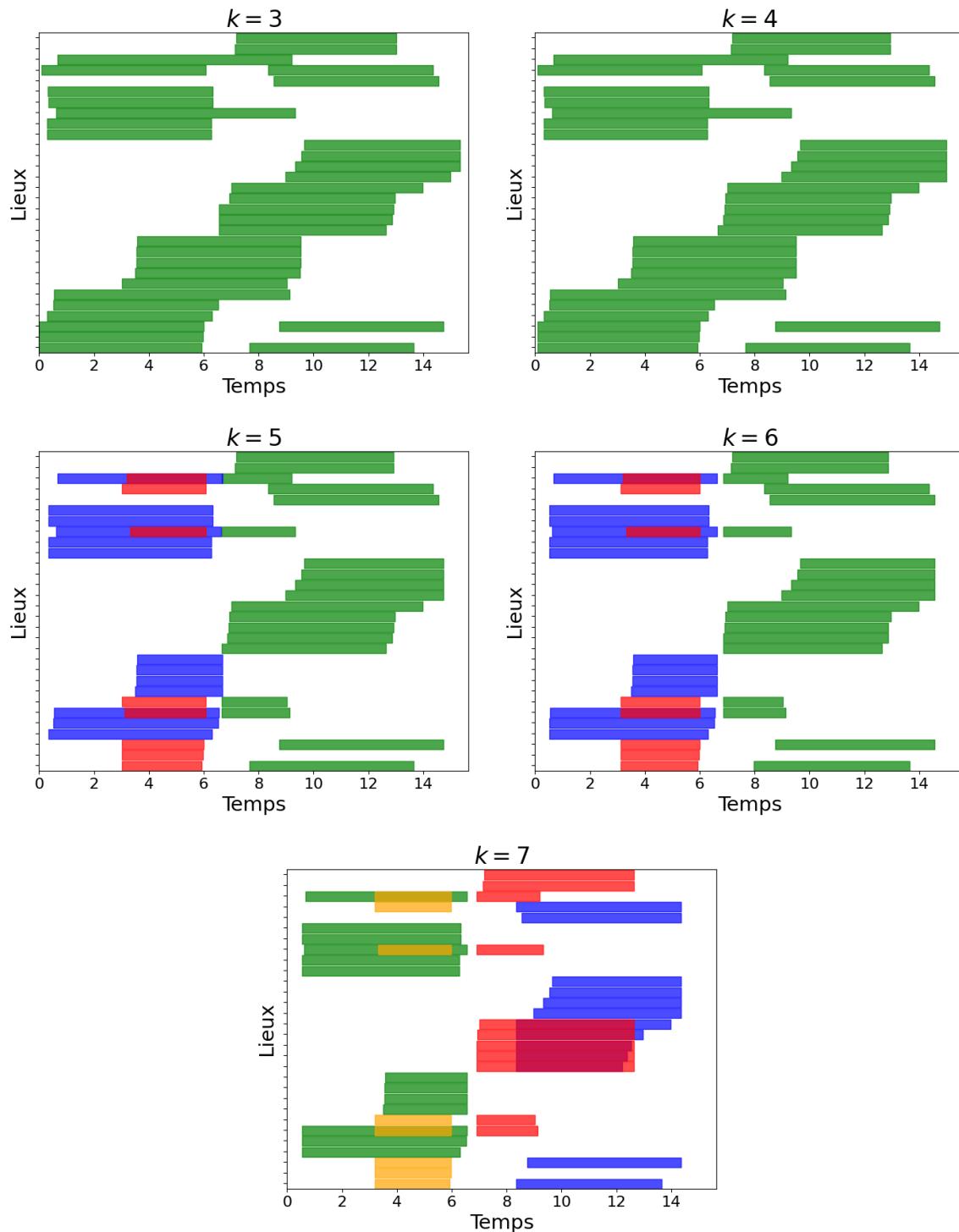


FIGURE 6.13 – Fragmentation d'une communauté de *Foursquare* lorsque k augmente de $k = 3$ à $k = 7$. Toutes les communautés incluses dans la communauté pour $k = 3$ sont représentées. Chaque couleur représente une communauté différente.

6.6.6 Mesurer le taux de dynamicité d'une communauté

L'algorithme LSCPM produit de nombreuses communautés, et pour les grands jeux de données, elles sont trop nombreuses pour être aisément explorées individuellement. Cela soulève la question de savoir comment les distinguer en fonction de la manière dont elles regroupent les sommets au fil du temps. Nous introduisons alors ce que nous appelons le ***taux de dynamicité*** d'une communauté. Intuitivement, lorsque chaque sommet est présent pendant toute la durée d'une communauté, celle-ci n'est pas dynamique. Au contraire, si une partie des sommets n'est présente que sur des durées bien plus courtes, cela montre que la communauté varie de manière plus dynamique.

Étant donnée une communauté C , nous définissons $t_d(C)$ le taux de dynamicité de C comme suit : $t_d(C) = 1 - \frac{\sum_{u \in C} \text{dur}_C(u)}{n_C \cdot D_C}$, où D_C est la durée de C , c'est-à-dire le temps écoulé entre l'apparition de son premier sommet et l'extinction de son dernier sommet, $\text{dur}_C(u)$ est la durée totale pendant laquelle le sommet u appartient à C , et n_C est le nombre de sommets dans C . Plus $t_d(C)$ est proche de 0, plus les sommets qui constituent C sont présents sur l'intégralité de son intervalle de temps d'existence ; plus $t_d(C)$ est proche de 1 et plus les sommets sont présents pendant de petites fractions de temps.

Par exemple, la Figure 6.10 illustre des communautés avec un fort taux de dynamicité. La représentation sous forme de flot de liens permet de donner un critère visuel pour classer ces communautés : les plus dynamiques sont celles dans lesquelles on peut observer une grande surface non remplie.

La figure 6.14 présente la distribution cumulative de ce taux, sur les communautés de nos jeux de données. On constate que chaque jeu de données possède au moins 40% de ses communautés qui sont totalement compactes (lorsque le taux est nul). Cela correspond dans la plupart des cas à des k -cliques qui sont adjacentes à aucune autre. On observe également des communautés ayant un taux de 0,8 pour *Infectious* et jusqu'à près de 1 pour *Wikipedia*. Entre ces valeurs extrêmes, il existe toute une gamme de valeurs intermédiaires qui méritent d'être étudiées. Ainsi, les communautés sont triées le long de la distribution cumulative, permettant de cibler les communautés les plus dynamiques pour étudier la temporalité des interactions. Par exemple, les communautés de la figure 6.10 font parties des communautés ayant le taux de dynamicité le plus élevé de leur jeu de données, avec un taux de 0.54 pour la communauté de *Households*, et un taux de 0.59 pour la communauté de *Hightschool*. On a vu qu'on pouvait extraire de l'information pertinente de leur dynamique, il est donc intéressant de pouvoir les cibler avec cet indicateur.

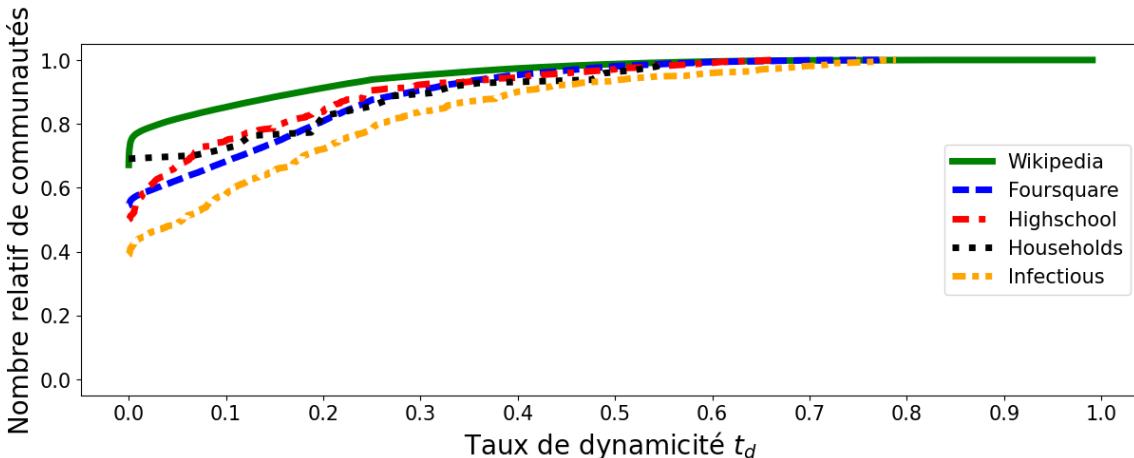


FIGURE 6.14 – Distribution cumulative du taux de dynamicité des communautés, pour chaque jeu de données.

6.7 Conclusion et perspectives

Dans ce chapitre, nous avons abordé la détection des communautés dynamiques dans les réseaux temporels. Grâce au formalisme des flots de liens, qui fournit une symétrie entre la structure des données et le temps, nous avons pu exploiter les algorithmes efficaces sur les graphes pour les étendre aux interactions temporelles. En nous appuyant sur la littérature dans ce domaine, nous avons introduit la notion de k -clique maximale d'un flot de liens, ainsi qu'un algorithme permettant de les énumérer. Cela conduit à une nouvelle adaptation de la méthode de percolation de cliques aux flots de liens, appelée LSCPM, qui poursuit les travaux initiés par Palla *et al.* [PBV07], en donnant une nouvelle définition des communautés dynamiques, plus agrégées dans le temps et avec un algorithme plus efficace pour les calculer. Nous avons fourni une analyse théorique de la complexité de notre algorithme, ainsi qu'une implémentation open source en Python. Ensuite, nous avons réalisé des expériences avec l'algorithme, en le comparant à l'état de l'art, et nous avons montré qu'il permet d'obtenir des informations pertinentes sur des exemples issus du monde réel.

Même si nous ne l'avons pas expérimentée ici, la consommation de mémoire peut être un facteur limitant sur les flots de liens massifs en raison du stockage des cliques. Nous pensons que la détection de communautés avec LSCPM peut s'étendre à des réseaux encore plus massifs et à de plus grandes valeurs de k . Le travail effectué dans le chapitre 3 pour réduire le coût en mémoire de la méthode CPM sur les graphes pourrait alors être adapté aux flots de liens. De plus, il pourrait être préférable dans certains cas de faire la percolation sur les cliques maximales à la place des k -cliques, comme cela est fait dans OCPM [Bou+18], en utilisant des méthodes efficaces d'énumération de cliques maximales dans des flots de liens tels que celle développée

dans le chapitre 5. En particulier, nous avons vu qu'il peut s'agir d'une alternative efficace pour des grandes valeurs k .

De plus, on a vu qu'il était possible de moduler les jeux de données en faisant varier la durée Δ de leurs liens. Il serait intéressant de développer l'analyse de l'effet de la modification de cette durée des liens Δ et de ses implications pratiques. En effet, lorsque Δ augmente, les k -cliques deviennent plus longues dans le temps, ce qui se traduit par des communautés LSCPM plus agrégées. C'est l'effet inverse de ce qui se produit lorsque l'on augmente k . Nous pensons que l'étude expérimentale peut être étendue en testant simultanément ces deux paramètres, afin de voir si cela permet de cibler les noyaux d'interaction pertinents en particulier.

Conclusion et perspectives

7.1 Contributions

L'objectif de cette thèse a été d'apporter de nouvelles réflexions sur l'étude des réseaux massifs issus du monde réel. Nous nous sommes concentrés sur les cliques, qui sont des sous-graphes importants pour apporter de l'information sur la structure de ces réseaux. Nous avons étudié l'énumération des cliques, ainsi que la détection de communautés qu'elles permettent de mettre en œuvre. Nous avons créé de nouveaux algorithmes pour répondre à deux objectifs principaux : étendre les résultats sur les réseaux statiques aux réseaux dynamiques afin d'avoir des outils efficaces pour les analyser, et de manière générale améliorer la performance des calculs. Nous nous sommes intéressés non pas au cas générique, mais à concevoir des algorithmes efficaces en pratique sur les graphes issus de contextes réels, en exploitant leurs propriétés particulières.

Dans chacun des chapitres 3, 4, 5 et 6, nous avons développé de nouveaux algorithmes, desquels nous avons fait une analyse de complexité théorique, ainsi qu'une série d'expériences qui valident les avancées qu'ils apportent, en fournissant des implémentations *open source*. En particulier, nous les avons comparés à ceux qui existent dans l'état de l'art : ils nous ont permis d'obtenir des solutions sur des réseaux massifs du monde réel pour lesquels aucun algorithme de l'état de l'art n'était en capacité de fournir un résultat. Dans le chapitre 3, cela est dû à un gain dans la mémoire nécessaire au calcul qui était un facteur limitant, et dans les chapitres 4, 5 et 6, cela est dû à une amélioration majeure du temps de calcul pour ces réseaux massifs.

Dans le chapitre 3, nous avons présenté un nouvel algorithme pour la détection de communautés par percolation de cliques dans les graphes. Notre contribution y est double. Tout d'abord, nous avons amélioré l'implémentation de l'état de l'art en tirant profit d'un des meilleurs algorithmes d'énumération des k -cliques dans les graphes. Ceci nous a permis d'en fournir une implémentation en C qui est bien plus rapide que celles de l'état de l'art. À partir de cette implémentation, nous avons fait le constat que c'est la mémoire qui est le facteur limitant sur les graphes massifs. Nous avons alors créé une nouvelle structure de données, que nous avons

appelé *Overlapping Union-Find*, qui est une généralisation de la structure de données *Union-Find* afin de faire l’union d’ensembles *non-disjoints*. Cette structure nous a permis de développer une définition moins stricte de la percolation de cliques, qui est moins coûteuse en mémoire. Nous avons alors introduit un nouvel algorithme qui fournit des communautés très proches de celles de l’état de l’art, en utilisant plus de temps de calcul, mais beaucoup moins de mémoire.

Dans le chapitre 4, nous avons développé un nouvel algorithme pour l’énumération des bicliques maximales dans les graphes bipartis. Pour cela, nous avons relevé une équivalence entre les bicliques d’un graphe biparti et les cliques de son graphe que l’on appelle étendu. Néanmoins, ce graphe étendu est trop dense pour que l’on puisse exécuter dessus l’algorithme classique d’énumération des cliques maximales dans les graphes. Nous avons alors proposé une méthode algorithmique qui permet d’énumérer les cliques maximales de ce graphe, mais en n’utilisant que le voisinage de son graphe biparti, et pas celui du graphe étendu. Cela nous a permis de profiter du caractère clairsemé des graphes bipartis massifs du monde réel et a abouti à un algorithme très efficace pour l’énumération des bicliques maximales. Dans nos expériences, nous avons obtenu un gain de temps de calcul d’au moins un facteur 10 sur les graphes contenant plus de dix millions de bicliques maximales, et cela nous a permis d’obtenir en quelques heures l’ensemble des bicliques maximales sur des graphes pour lesquels l’état de l’art ne termine pas dans la limite d’une semaine de calcul. Nous avons illustré le fait que l’ordre des sommets et le choix de l’un des deux ensembles du graphe biparti sur lequel l’énumération est exécutée ont un impact sur le temps de calcul. En particulier, nous avons constaté que le comportement de l’algorithme n’était pas le même sur les graphes les plus massifs et sur les petits graphes de nos jeux de données, ce qui donne une clé de compréhension des graphes bipartis du monde réel.

Le chapitre 5 a amélioré l’énumération des cliques maximales dans les flots de liens, qui modélisent les réseaux temporels. La contribution algorithmique majeure qui nous a permis d’améliorer le temps de calcul a été de faire un parallèle théorique entre les cliques du flot de liens et les cliques de ses graphes instantanés (graphes des interactions qui existent à chaque instant). Cela nous a permis de ne travailler qu’avec des opérations au sein de ces graphes instantanés, et donc d’exploiter le caractère clairsemé de ces réseaux, sans avoir à traiter les interactions temporelles dans leur intégralité. Nous en avons fait une étude théorique complète qui a montré la validité de cette extension et nous avons établi deux formules de complexité : une en fonction de l’entrée de l’algorithme et une en fonction des caractéristiques de la sortie. Cette dernière permet de voir dans quelle mesure l’algorithme est proche d’une énumération optimale, qui est au moins de la taille de la sortie. Nos expériences ont montré un gain de temps de calcul de plusieurs ordres de grandeur par rapport aux algorithmes de l’état de l’art.

Enfin, dans le chapitre 6, nous avons développé une nouvelle définition de com-

munautés dans les flots de liens, qui est une extension directe de l'algorithme de percolation de cliques dans les graphes étudié dans le chapitre 3. Nous avons tout d'abord défini une extension des k -cliques dans ce contexte temporel, et nous avons développé un algorithme pour les énumérer. Pour mettre en œuvre la percolation des cliques, nous avons introduit la structure de *Temporal Union-Find*, pour faire des opérations d'union très efficaces entre des ensembles d'éléments temporels (chaque élément est associé à un ou plusieurs intervalles de temps disjoints pendant lesquels il appartient à l'ensemble). Nous avons comparé cet algorithme à l'extension de la percolation de cliques qui a été proposée pour les graphes temporels dans la littérature. Notre nouvelle définition donne des communautés proches de celles de l'état de l'art et le temps de calcul pour les obtenir est meilleur de plusieurs ordres de grandeur. Nous avons proposé une étude expérimentale de notre algorithme, qui montre des exemples d'information pertinente que ces communautés temporelles permettent d'obtenir sur des jeux de données issus du monde réel.

7.2 Perspectives

Les avancées réalisées au cours de cette thèse ouvrent la voie à plusieurs perspectives de recherche.

Vers une amélioration de l'énumération des cliques maximales dans les graphes. Dans le chapitre 4, nous avons présenté l'algorithme d'Eppstein *et al.*, qui est l'un des meilleurs algorithmes d'énumération des cliques maximales dans les graphes massifs du monde réel. Cette méthode implique d'énumérer, pour chaque sommet, les cliques maximales qui le contiennent et qui n'ont pas encore été énumérées. Ce processus repose sur un ordre judicieux sur les sommets, qui sont traités selon leur dégénérescence croissante. Lorsqu'il s'agit d'un graphe biparti $G = (U, V, E)$, nous avons vu qu'on pouvait choisir de travailler à partir de l'un des ensembles de sommets, U ou V . Une observation intéressante a émergé de cette étude lors de nos expériences : pour les graphes massifs de notre jeu de données, il semble plus efficace de procéder à partir de l'ensemble contenant le plus de sommets, tandis que pour les autres, l'énumération est en général plus rapide à partir de l'ensemble contenant le moins de sommets. Nous avons vu que cela provenait du fait que, même si nous réalisons l'énumération plus fréquemment lorsque le nombre de sommets est plus élevé, si les ensembles de sommets à parcourir pour faire grossir la clique en construction contiennent moins de sommets, cela peut entraîner une réduction significative du temps de calcul.

Cette observation conduit directement à une perspective algorithmique sur l'énumération des cliques maximales. Nous proposons d'explorer l'idée d'itérer les appels initiaux sur les *arêtes* du graphe au lieu des sommets. À première vue, cela peut sembler ne pas apporter de bénéfice particulier, car cela revient essentiellement à

partir des sommets et à ne pas réaliser de pivot au sein de chaque premier appel récursif. Cependant, pour chaque arête, l'ensemble de sommets à parcourir pour construire les cliques la contenant sont les voisins communs des deux sommets de cette arête. D'une part, cet ensemble est en général plus petit que l'ensemble des voisins d'un sommet. D'autre part, les arêtes pourraient être ordonnées en leur étendant le concept de dégénérescence, en leur définissant un voisinage qui serait l'intersection des voisinages de leurs deux sommets. Cet ordre pourrait aboutir à des ensembles de recherche bien plus petits. Alors, bien que cela fasse augmenter le nombre de sous-problèmes d'énumération par rapport à la méthode par sommet, leur résolution serait accélérée par le fait qu'ils sont de plus petite taille. Cela pourrait ainsi se traduire par une amélioration globale du temps de calcul.

Cette approche mérite d'être formalisée et expérimentée, en particulier sur les graphes massifs, et notamment sur ceux où l'énumération est actuellement bloquante, afin de voir s'il permet de mieux les traiter.

Relaxer la définition des cliques pour l'étude des communautés. Dans cette thèse, notre attention s'est principalement portée sur l'étude des cliques. Toutefois, il est important de noter qu'il existe d'autres définitions de sous-graphes denses qui permettent une certaine flexibilité, notamment en autorisant quelques liens à manquer au sein de la clique. C'est le cas par exemple des k -plexes, qui sont des ensembles de sommets qui autorisent les sommets à ne pas être connecté à au plus k autres sommets de l'ensemble, par rapport aux cliques qui imposent aux sommets d'être connectés à tous les autres. Cette flexibilité s'avère particulièrement pertinente pour l'analyse de réseaux réels, dans lesquels des arêtes peuvent être manquantes du fait de l'enregistrement des données, ou pour lesquels des regroupements denses ne forment pas nécessairement des cliques parfaites. De plus, cette notion offre des opportunités pour la prédiction des liens manquants dans les graphes. Cela soulève des considérations algorithmiques spécifiques qui méritent d'être approfondies.

En parallèle, la structure des k -plexes peut jouer un rôle essentiel dans le développement d'une nouvelle notion de communautés, plus flexible. Contrairement à la percolation de cliques, qui exige qu'un sommet appartienne à une clique d'au moins k membres pour faire partie d'une communauté, l'ensemble des k -plexes peut englober un plus grand nombre de sommets. L'idée de définir des communautés par la percolation de k -plexes mérite donc d'être explorée pour évaluer son utilité dans l'analyse de données réelles. De plus, cela soulève des questions algorithmiques intéressantes, en ce qui concerne l'énumération de ces structures, qui est actuellement moins efficace que pour les cliques, et la règle d'adjacence serait à formaliser.

Étudier l'impact de la durée dans les flots de liens. Dans les chapitres 5 et 6, nous avons abordé la modélisation des réseaux temporels en utilisant des flots

de liens dans lesquels les interactions ont une durée. Cette approche permet de traiter des interactions temporelles, comme des interactions physiques ou des appels téléphoniques. Cependant, il existe également des données temporelles où les interactions se produisent de manière instantanée, sans durée associée, comme c'est le cas des échanges de messages. Dans de tels cas, nous avons vu que nous pouvions toujours appliquer les algorithmes de flots de liens avec durée en introduisant une durée Δ pour ces interactions. Par exemple, si nous fixons une durée Δ d'une semaine à des échanges de mails entre utilisateurs, une clique entre ces utilisateurs représente un ensemble d'interactions se produisant dans une fenêtre de temps d'au plus une semaine. De plus, dans certains cas, cette durée permet d'inférer des interactions existantes. Par exemple, en mesurant des interactions de proximité physique par des capteurs, on peut interpréter plusieurs interactions entre deux mêmes personnes mesurées dans des intervalles de temps inférieurs à Δ comme une interaction continue entre ces deux personnes.

Il est essentiel de comprendre que l'ajout et la modification de cette durée Δ ont un impact sur la structure sous-jacente du flot de liens. L'augmentation de Δ tend à réduire le nombre de liens en fusionnant ceux qui existent entre deux mêmes sommets lorsqu'ils se chevauchent dans le temps une fois la durée Δ ajoutée. Elle peut également rendre le réseau plus dense du fait que les liens durent plus longtemps dans le temps. De même, le nombre de cliques peut diminuer, car celles qui partagent les mêmes sommets fusionnent lorsqu'elles se chevauchent dans le temps une fois la durée Δ ajoutée. Mais leur nombre peut aussi augmenter, car des sommets qui ne sont pas en relation pour de petites valeurs de Δ peuvent le devenir lorsque Δ augmente. Cette observation a notamment été étudiée par Viard *et al.* [VML18] et Himmel *et al.* [Him+17] dans le contexte de leur algorithme d'énumération des cliques maximales. De plus, nous avons remarqué que dans notre algorithme, le temps d'énumération et le nombre de cliques varie en fonction de la valeur de Δ , avec des temps d'énumération plus longs pour les valeurs de Δ les plus élevées. Il serait donc important de mener une analyse qui évaluerait l'influence de Δ sur les paramètres structurels du flot de liens et l'impact de cette durée sur le comportement des algorithmes face aux différents réseaux.

Poursuivre l'exploration des graphes massifs issus du monde réel. Comme nous l'avons déjà constaté, de nombreux algorithmes affichent des performances bien supérieures à leurs pires cas théoriques lorsqu'ils sont appliqués à des graphes du monde réel. Comprendre pourquoi ces algorithmes fonctionnent si bien sur ces instances demeure une question ouverte, qui n'est pas simplement expliquée par des formules de complexité. Par exemple, dans le cas de l'énumération des cliques maximales, il serait intéressant d'étudier plus en détail les graphes pour lesquels l'énumération ne termine pas dans un délai raisonnable, afin d'en comprendre la cause précise, qui, une fois identifiée, pourrait déboucher sur des algorithmes qui en tireraient profit. En particulier, deux cas peuvent déjà être distingués : les énuméra-

tions qui ne terminent pas parce qu'il y a trop d'éléments à énumérer, et celles qui ne terminent pas, car c'est le temps de traitement du graphe qui est trop long en lui-même. Dans nos travaux, nous nous sommes particulièrement intéressé au caractère *clairsemé* de ces graphes, ou encore à leur *faible dégénérescence*. Cependant, il existe encore de nombreuses autres propriétés spécifiques aux graphes du monde réel qui peuvent être identifiées et exploitées. Par exemple, des études ont travaillé sur la somme des carrés des degrés qui est faible en pratique, et différents ordres sur les sommets pour raffiner cette valeur [Léc+23], tout comme l'ordre par dégénérescence a permis de raffiner la prise en compte du degré des sommets dans l'énumération des cliques maximales. Nous pensons que la recherche de ces propriétés existantes, ainsi que la découverte de nouvelles, pourrait grandement profiter au développement de nouveaux algorithmes.

Il est essentiel de noter que même en exploitant des propriétés spécifiques des graphes réels, la conception d'algorithmes exacts peut s'avérer complexe et leur exécution impossible sur des données massives. C'est pourquoi nous envisageons également d'explorer la voie des algorithmes d'approximation qui peuvent fournir des solutions acceptables dans des délais raisonnables.

Enfin, pour traiter des jeux de données massifs, de plus en plus d'algorithmes parallélisables sont développés. Cela permet naturellement de diviser leur temps de calcul par (au plus) le nombre de *threads* sur lesquels ils sont exécutés. Il est néanmoins important de noter que tous les algorithmes ne sont pas naturellement parallélisables, et certains peuvent nécessiter des modifications substantielles pour tirer pleinement parti d'une architecture parallèle. Ainsi, développer des algorithmes qui sont parallélisables dans nos futurs travaux participerait à améliorer la recherche autour des graphes massifs du monde réel.

Vers une méthode pour étendre les algorithmes sur graphes à des structures plus générales ? Nous avons élaboré de nouveaux algorithmes destinés non seulement aux graphes classiques, mais également aux graphes bipartis et aux flots de liens. Notre approche a consisté à établir des parallèles entre ces différentes structures et les graphes classiques, nous permettant ainsi de concevoir des algorithmes en y adaptant des méthodes déjà existantes pour les graphes. Une perspective intéressante serait d'explorer la mise en place de procédures de généralisation globales, permettant d'adapter un grand nombre d'algorithmes à ces différentes structures de manière plus systématique.

Pour illustrer cette idée, prenons l'exemple du graphe étendu que nous avons introduit pour les graphes bipartis dans le chapitre 4. On pourrait appliquer à ce graphe étendu des algorithmes conçus pour les graphes classiques, afin d'obtenir des algorithmes directement pour les graphes bipartis suivant la même logique, ou bien définir d'autres types de graphes étendus. De même, dans le domaine des flots de liens, nous pourrions envisager l'adaptation de l'algorithme d'énumération des k -

cliques présenté dans le chapitre 6 à un cadre plus général d'énumération de motifs temporels. En effet, nous avons constaté que l'appel à l'algorithme que nous utilisons pour énumérer les cliques dans les graphes des liens agrégés, pourrait être remplacé par n'importe quel autre algorithme de recherche de motifs dans un graphe, générant ainsi directement des motifs temporels. Idéalement, nous pourrions proposer une trame d'algorithme générale dans laquelle l'utilisateur entre le motif à généraliser et une méthode sur les graphes pour l'obtenir, et elle renvoie les motifs temporels associés. Cette approche ouvre la voie à la création de nouveaux algorithmes efficaces pour la recherche de motifs dans les flots de liens.

En outre, selon le contexte, nous pourrions exploiter des structures plus complexes que les graphes traditionnels, telles que les *hypergraphes* (où les arêtes peuvent regrouper plus de deux sommets) ou les *graphes multicouches* (comportant plusieurs ensembles d'arêtes). En nous inspirant de notre extension des algorithmes sur graphes aux graphes bipartis et aux flots de liens, nous pourrions envisager la création de nouveaux algorithmes pour résoudre des problèmes clés dans les hypergraphes et les réseaux multicouches, en adaptant les outils conçus pour les graphes. Dans l'ensemble, notre objectif serait de développer une théorie permettant d'établir des liens entre les résultats obtenus dans une structure donnée et leur application potentielle dans d'autres structures similaires, favorisant ainsi une approche plus unifiée de l'analyse des réseaux complexes.

Vers une ouverture interdisciplinaire. Les graphes, en tant que représentation abstraite des relations entre des entités, offrent un langage universel pour décrire et analyser des réseaux complexes. La théorie des graphes a ainsi une capacité à dépasser les frontières disciplinaires pour aborder des problèmes interconnectés. Cette universalité permet aux chercheurs de différentes disciplines de dialoguer et de collaborer en utilisant un cadre commun. Travailler avec des données réelles d'un domaine spécifique et comprendre leur structure permet de trouver et reconnaître des spécificités dans ces réseaux, utiles à leur analyse. Par exemple, récemment, il a été mis en évidence une propriété intéressante sur les réseaux d'interactions protéine-protéine : Kovacs *et al.* [Kov+19] ont observé que les chemins de longueur 3 étaient de bons prédicteurs d'interactions en se basant sur l'intuition que si une protéine de type A interagit avec une protéine de type B, alors on pourrait trouver des motifs correspondant à des interactions entre toutes les protéines A et toutes les protéines B, c'est-à-dire des bicliques. Ils utilisent alors ces structures de bicliques pour prédire de nouvelles interactions, en complétant les quasi-bicliques avec des liens prédictifs. D'autres travaux cherchent depuis à raffiner cette méthode [YJ23]. Pour accomplir cela efficacement, ils ont besoin d'un algorithme capable d'énumérer des quasi-bicliques, dans des graphes qui ne sont pas bipartis, ouvrant ainsi un nouveau problème algorithmique. Par ailleurs, le dénombrement de motifs dans les graphes a des applications importantes en apprentissage. Certaines applications, comme la détection de fraudes dans les transactions financières, demandent de détecter

ter des anomalies ou classer des instances à la volée, ce qui nécessite des algorithmes efficaces de détection de motifs, qui sont difficiles à mettre en œuvre.

Prenons un exemple plus général : le cerveau est un système complexe composé de multiples éléments qui interagissent entre eux. Plusieurs études ont montré le potentiel de collaboration entre les communautés des sciences des réseaux et des neurosciences [Bar+23 ; BS17], pour étudier les transitions neuronales du développement des fonctions saines ou de maladies. Deux tendances parallèles stimulent cette approche : la disponibilité de nouveaux outils empiriques pour mesurer et enregistrer des réseaux statiques ou dynamiques parmi les molécules, les neurones, les aires cérébrales ; et le cadre théorique de la science moderne des réseaux accompagné de ses outils computationnels. La convergence des avancées empiriques et computationnelles ouvre de nouveaux horizons pour les neurosciences, par exemple en utilisant la dynamique des réseaux [Gao+11 ; RDN23], la manipulation et le contrôle des réseaux cérébraux [Bas+21], ou encore la combinaison des domaines structurels et fonctionnels des interactions cérébrales grâce à l'étude des réseaux multicouches [PF22].

Pour citer un autre exemple majeur, en écologie, plusieurs études soulignent la nécessité d'avoir une image complète des interactions écologiques et de leur variabilité géographique pour prédire les effets des changements environnementaux sur les écosystèmes. Des chercheurs étudient la manière dont l'ensemble des interactions écologiques entre les espèces au sein de communautés complexes, varient dans le temps et à travers des étendues géographiques [Lur+20]. Formaliser et étudier des notions comme la robustesse d'un réseau écologique permet de quantifier la résilience de l'écosystème qu'il représente face à la disparition d'espèces. Dans cet exemple, le lien entre la structure du réseau et la robustesse reste une question ouverte, et nécessite le développement d'une théorie particulière [CBD22].

Ainsi, l'utilisation des graphes est un enjeu central dans de nombreuses problématiques contemporaines. L'étude des graphes issus du monde réel semble être un élément important pour résoudre des problèmes algorithmiques actuels et futurs. Collaborer au-delà des frontières traditionnelles des disciplines pourrait repousser les limites de la connaissance et contribuer de manière significative à la résolution des problèmes les plus pressants de notre époque.

Collaboration interdisciplinaire

Pendant la thèse, un partenariat a été réalisé avec le LCQB (*Laboratory of Computational and Quantitative Biology*) qui est un laboratoire interdisciplinaire à l’interface entre la biologie et les sciences quantitatives, à Sorbonne Université. Nous avons travaillé sur un problème de bioinformatique de comparaison de séquences biologiques, à travers une modélisation sous forme de graphes, appelés *graphes d’épissage évolutifs* (ESG). L’approche a été implémentée dans *ThorAxe*, un outil de calcul disponible en ligne¹.

Plus précisément, l’objectif de ce travail a été de construire un graphe associé à n gènes g_1, \dots, g_n . Ces gènes sont des versions d’un même gène, chacun pour une espèce différente, que l’on veut comparer pour montrer les impacts que l’évolution a eus sur ce gène. La représentation sous forme de graphe permet d’exhiber les différences et les similarités entre ces différentes versions.

Notre contribution a été de démontrer que cette construction est NP-difficile, et d’en rédiger une démonstration, pour une publication dans le journal *Genome Research* [Zea+21]. Pour cela, nous en avons fait une réduction depuis un problème d’alignement de séquences multiples avec score de somme de paires, qui est un problème NP-difficile [WJ94]. Nous mettons la publication dans cet annexe, à la page suivante. La démonstration se trouve à la page numérotée 1470 dans l’article.

Ce travail est un exemple de l’importance de l’étude interdisciplinaire des graphes.

1. <https://github.com/PhyloSofS-Team/thoraxe>

Method

Assessing conservation of alternative splicing with evolutionary splicing graphs

Diego Javier Zea,¹ Sofya Laskina,² Alexis Baudin,³ Hugues Richard,^{1,2} and Elodie Laine¹

¹Sorbonne Université, CNRS, IBPS, Laboratoire de Biologie Computationnelle et Quantitative (LCQB), 75005 Paris, France;

²Bioinformatics Unit (MF1), Department for Methods Development and Research Infrastructure, Robert Koch Institute, 13353 Berlin, Germany; ³Sorbonne Université, CNRS, LIP6, F-75005 Paris, France

Understanding how protein function has evolved and diversified is of great importance for human genetics and medicine. Here, we tackle the problem of describing the whole transcript variability observed in several species by generalizing the definition of splicing graph. We provide a practical solution to construct parsimonious evolutionary splicing graphs where each node is a minimal transcript building block defined across species. We show a clear link between the functional relevance, tissue regulation, and conservation of alternative transcripts on a set of 50 genes. By scaling up to the whole human protein-coding genome, we identify a few thousand genes where alternative splicing modulates the number and composition of pseudorepeats. We have implemented our approach in ThorAxe, an efficient, versatile, robust, and freely available computational tool.

[Supplemental material is available for this article.]

Eukaryotes have evolved a transcriptional mechanism that can augment the protein repertoire without increasing genome size. A gene can be transcribed, spliced, and matured into several transcripts by choosing different initiation/termination sites or by selecting different exons (Graveley 2001). Alternative splicing, as well as alternative promoter usage or alternative polyadenylation (hereafter referred to as AS), concerns almost all multiexon genes in vertebrates (Wang et al. 2008), and many organ-specific splicing patterns have diverged rapidly during vertebrate evolution (Barbosa-Morais et al. 2012; Merkin et al. 2012). This mechanism can affect transcript maturation and post-transcriptional regulation or result in protein isoforms ("proteoforms") with different shapes (Birzele et al. 2008), interaction partners (Yang et al. 2016), and functions (Kelemen et al. 2013; Baralle and Giudice 2017). Its misregulation is associated with the development of cancer, among other diseases (Wang and Cooper 2007; Ward and Cooper 2010; Lim et al. 2011; Scotti and Swanson 2016; Clemente-González et al. 2017). Moreover, the influence of natural isoform variations between human populations on disease susceptibility is increasingly recognized (Park et al. 2018). Hence, understanding how AS contributes to protein function diversification is of utmost importance for human genetics and medicine.

In recent years, the advent of high-throughput sequencing technologies like RNA-seq has made possible in-depth surveys of transcriptome complexity (Sultan et al. 2008; Wang et al. 2008). However, evaluating how many of the detected transcripts are translated and functional in the cell remains challenging (Wang et al. 2018). This difficulty has stimulated the development of integrative approaches combining gene annotations, RNA-seq data, and also data generated by other high-throughput techniques (De La Grange et al. 2010; González-Porta et al. 2013; Rodriguez et al. 2013; Ezkurdia et al. 2015; Weatheritt et al. 2016; Tapial et al. 2017; Tranchevent et al. 2017; Denti et al. 2018; Sterne-Weiler et al.

2018; Agosto et al. 2019; Ait-hamlat et al. 2020; de la Fuente et al. 2020; Louadi et al. 2020; Martí-Solano et al. 2020) toward a better characterization of the AS landscape. Recent studies underscore AS functional impact and contribution to protein diversity (Agosto et al. 2019; Martí-Solano et al. 2020).

Evolutionary conservation can arguably serve as a reliable indicator of function. Indeed, we expect splice variants selected over millions of years of evolution to comply with physical and environmental constraints and thus to play a functional role. The classical approach for assessing AS evolutionary conservation first identifies orthologous exons between species and then compares their inclusion/exclusion rates across cell/tissue types. A common practice for orthology detection is the BLAST (Altschul et al. 1990) "all-against-all" methodology (Nichio et al. 2017). When dealing with exons, more specialized protocols based on pairwise genomic sequence alignments (Modrek and Lee 2003; Xing and Lee 2005; Barbosa-Morais et al. 2012; Abascal et al. 2015; Herrero et al. 2016; Mei et al. 2017) or multiple alignments of genomic or protein sequences (Christinat and Moret 2012; Merkin et al. 2012; Szalkowski 2012) have been proposed. Challenges associated with this task include correctly handling large indel events, finding plausible matches for highly divergent sequences, and resolving ambiguities arising from highly similar sequences (e.g., resulting from in-gene duplication) or very short sequences. The alternative usage of orthologous exons may then be investigated using compact representations of transcript variability, such as splicing graphs (Heber et al. 2002), where the nodes represent the exons, and the edges denote exon junctions. Hence, a way to assess AS conservation between two genes would be to compare the environment of their orthologous exons in the two corresponding splicing graphs. However, until now, there exists no method combining these two layers of information.

Corresponding authors: RichardH@rki.de,
elodie.laine@sorbonne-universite.fr

Article published online before print. Article, supplemental material, and publication date are at <https://www.genome.org/cgi/doi/10.1101/gr.274696.120>.

© 2021 Zea et al. This article is distributed exclusively by Cold Spring Harbor Laboratory Press for the first six months after the full-issue publication date (see <https://genome.cshlp.org/site/misc/terms.xhtml>). After six months, it is available under a Creative Commons License (Attribution-NonCommercial 4.0 International), as described at <http://creativecommons.org/licenses/by-nc/4.0/>.

Evolutionary splicing graphs for AS conservation

In this work, we addressed the problem of exon orthology detection in the context of AS. Our specific aims were to develop a novel and general method revisiting splicing graph representation to account for the whole transcript variability observed in a set of species and to apply this method at the protein-coding genome scale to provide, for the first time, granular estimates of AS evolutionary conservation and significantly improve our knowledge on the amount of variations that are functionally relevant.

Results

Evolution-informed model describes transcript variability

Our method maps all transcriptomic information coming from many species on an *evolutionary splicing graph*, where the nodes represent minimal transcript building blocks defined across species (Fig. 1). Classically, a splicing graph (SG) contains nodes representing exons, that is, genomic intervals, and edges indicating co-occurrences of contiguous exons in a set of transcripts observed for a gene. In the present work, we use a slightly different definition, in which the nodes are the genomic intervals supplemented by their reading frames (Supplemental Methods). In practice, we work with the corresponding translated amino acid sequences (Fig. 1A). Moreover, the nodes may actually represent *subexons*, because donor or acceptor sites can be located inside an exon

(Fig. 1A, n_1 , and n_2). We distinguish the edges *induced* by the sub-exon boundaries (Fig. 1A, $n_1 \rightarrow n_2$) from the *structural* edges arising from intron boundaries (Fig. 1A, $n_1 \rightarrow n_3$).

Our main contribution is to extend the definition of a splicing graph to a set of orthologous genes G . We describe the whole transcript variability of G with an evolutionary splicing graph (ESG) $\mathcal{S} = (\mathcal{V}, \mathcal{E})$ (Fig. 1B), where each node $v \in \mathcal{V}$ is a *spliced-exon* (*s-exon*) and represents a multiple sequence alignment (MSA) of subexons or subexon parts coming from different species. The edge set \mathcal{E} comprises the ensemble of edges linking the nodes in the individual SGs, possibly augmented by some edges induced by the definition of the *s-exons* (for more formal definitions, see Supplemental Methods). As a consequence, two nodes in \mathcal{S} may be linked by several edges (at most one per species), and we designate this set as a *multiedge*. There are many possible ways of grouping the exonic sequences coming from the different genes, and hence of defining the *s-exons* (Fig. 1B). For instance, one may define each *s-exon* in the ESG by taking at most one subexon from each SG (Fig. 1B, solutions 1 and 3). In that case, the edge set is exactly the set of edges coming from the individual SGs. Alternatively, it may be advantageous to split a subexon into two or more subsequences and assign them to different *s-exons* (Fig. 1B, solution 2, in which AEI and GV come from the human sub-exon AEIGV). This strategy can lead to better MSAs, but it increases the complexity of the graph. Ideally, one would like to find a

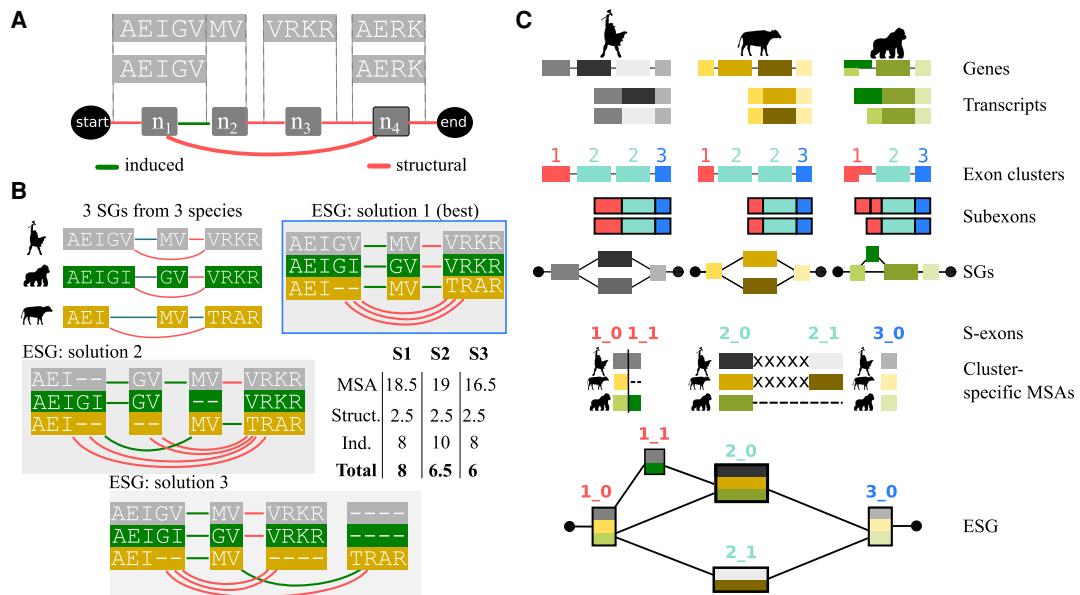


Figure 1. Principle of the method. (A) Two transcripts are depicted, in which each gray box represents a genomic interval and contains the corresponding protein sequence. (Below) The minimal SG is shown, with the nodes (n_1, n_2, n_3, n_4) corresponding to subexons. The start and end nodes are added for convenience. Each structural edge in red corresponds to some intron, and each induced edge in green corresponds to a junction located inside the initial genomic interval (such as the donor site of exon AEIGV). (B) Close-up view of three SGs corresponding to three orthologous genes coming from human, gorilla, and cow, along with three examples of ESGs summarizing the same information. The nodes in the ESGs represent *s-exons*, or multiple sequence alignments (MSAs) of exonic regions. The details of the ESG scores, computed from Equation 1, are given in the inset table, with $\sigma_{\text{match}} = 1$, $\sigma_{\text{mismatch}} = -0.5$, and $\sigma_{\text{gap}} = 0$ for the MSA scores, and edge penalties $\sigma_s = 0.5$ and $\sigma_t = 2$. The best-scored ESG shows at the same time compactness (parsimony) and good-quality alignments. (C) Main steps of the ESG construction in ThorAxe. The input genes and transcripts are depicted on top, with exons displayed as boxes. ThorAxe first step consists in grouping similar exons together. Here, three clusters are identified, colored in red (1), cyan (2), and blue (3); note that cluster 2 groups to multiple exons in human and cow. Then, subexons are defined based on intra-species transcript variability. For instance, the first exon from gorilla is split into two subexons. The subexons would be the nodes in the species-specific minimal SGs, although the latter are not explicitly computed by ThorAxe. The next step consists in aligning the sequences belonging to each cluster (with some padding "X" between mutually exclusive subexons) and identifying the spliced exons (*s-exons*) as blocks in the alignment. We keep track of the cluster IDs in the *s-exon* IDs, to ease interpretability. Finally, ThorAxe builds an ESG in which the nodes are the *s-exons*. For the sake of clarity, multiedges are visualized as single edges.

representation as compact as possible and at the same time, conveying meaningful evolutionary information. To estimate both properties, we define the score of the ESG S as

$$\sigma_{ESG}(S) = \sum_{v \in V} \sigma(v) - \sum_{e \in E} (n_e^I \cdot \sigma_I + n_e^S \cdot \sigma_S), \quad (1)$$

where $\sigma(v)$ is the score of the MSA associated to the node v . σ can be, for instance, a consensus score or a sum-of-pairs score and may additionally penalize very short MSAs (fewer than three columns). n_e^I (respectively, n_e^S) are the numbers of induced (respectively, structural) edges in the multiedge e with associated fixed penalty σ_I (respectively, σ_S). In practice, we set $\sigma_I \gg \sigma_S$ to avoid small s-exons induced by ambiguous alignment columns in the MSAs. As an example, with a simple sum-of-pair scoring for σ and an induced edge penalty $\sigma_I = 4 \cdot \sigma_S$, the best-scored ESG in Figure 1B (solution 1) comprises the smallest numbers of s-exons, induced edges, and gaps. In general, determining the best-scored ESG is a NP-hard problem (Methods).

Here, we provide a practical solution to construct a meaningful parsimonious ESG given a set of input transcripts (Fig. 1C; Supplemental Fig. S1). Our heuristic procedure first preclusters exons using pairwise alignments. Then, within each cluster, it concatenates the sequences coming from each species in the order of their genomic coordinates and aligns the obtained sequences using ProGraphMSA (Supplemental Fig. S2; Szalkowski 2012). The latter allows better handling of AS-induced deletions and insertions than classical progressive alignment methods. Moreover, using the genomic coordinates as ordering constraints helps to disentangle orthology from paralogy relationships between similar sequences (Supplemental Fig. S2C). Finally, we locally solve the problem exposed in Equation 1 by realigning some sequences and by maximizing the agreement between subexon boundaries

across different species (Supplemental Fig. S3). Controlling the creation of (penalizing) induced edges allows us to implicitly evaluate the ESG score. We implemented the heuristic in the fully automated tool ThorAxe.

In the following, we show that ThorAxe allows obtaining simple and meaningful representations for evolution in the context of AS. We primarily rely on gene annotations from Ensembl (Yates et al. 2016), and we complement the computed ESG with RNA-seq data from the NCBI Sequence Read Archive (SRA) (Leinonen et al. 2011), together with the tissue annotations compiled from Bgee (Supplemental Methods; Komljenovic et al. 2016). We focus on one-to-one orthologous genes across 12 species, namely three primates, two rodents, four other mammals, one amphibian, one fish, and nematode. Our motivation for this choice was to span different evolutionary distances and to ensure that enough RNA-seq data would be available. We take human as reference for selecting the genes, but ThorAxe ESG construction is reference-free.

ThorAxe recapitulates known functional AS events

We tested ThorAxe on a curated set of 50 genes representing 16 families (Supplemental Table S1; Supplemental Methods), in which several splice variants have been associated with diverse protein functions. ThorAxe detected 448 alternative splicing, initiation, and termination events. RNA-seq splice junctions mapping onto the ESGs provided additional support for about one-quarter of them and uncovered 101 more events. Detailed information is available on the accompanying website (<http://www.lcqb.upmc.fr/ThorAxe>) (Supplemental Fig. S4). We report here the results for a set of 30 documented events influencing partner binding affinity, selectivity, or specificity (Supplemental Tables S2, S3). We observed tissue-regulation patterns well-conserved across mammals for most of them; in amphibians, for seven of them (Fig. 2A; see also Supplemental Table S3). Although the gene annotations and the RNA-seq data show a good overall agreement, many subpaths are contributed solely by RNA-seq in platypus, cow, and zebrafish (Fig. 2B).

The ESG computed for *CAMK2B* linker region provides an illustrative example for which, despite a very high AS-generated complexity, ThorAxe results are interpretable, meaningful, and consistent with what has been reported in the literature (Fig. 3A). For instance, one can readily see that the shortest isoform lacking the linker (Fig. 3B, "7") has low evolutionary support. This is in line with recent findings emphasizing the importance of the linker for regulating the protein activity (Bhattacharyya et al. 2020). Moreover, all the s-exons defined by ThorAxe are conserved at least as far as amphibians. The smallest s-exon (25_1) contains only one column of alanines and corresponds to a well-documented internal splice site (Sloutsky and Stratton 2020). Finally, the two documented functional AS events are clearly identifiable on the ESG (Fig. 3A, gray areas). This observation still holds true

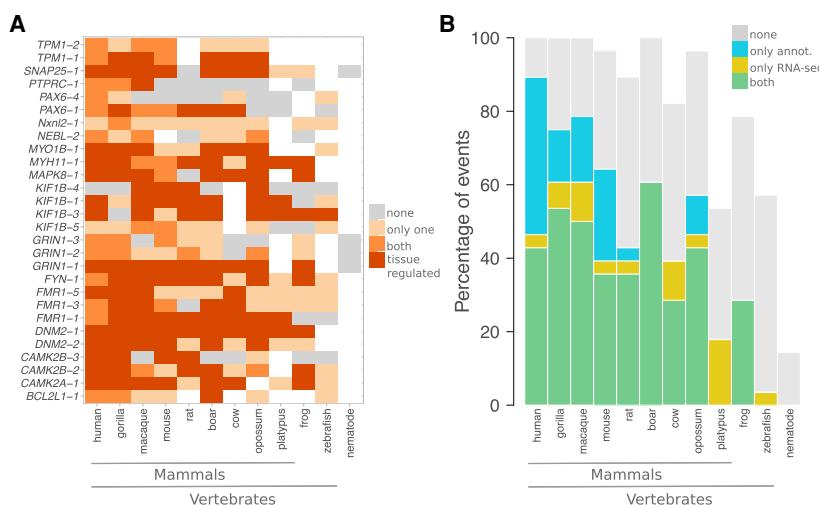


Figure 2. Conservation and tissue regulation of a set of documented AS events. (A) Each event is designated by the name of the gene where it occurs and its rank in ThorAxe output, the latter reflecting its relative conservation level. In the ESG, an event corresponds to a pair of subpaths, one being canonical and the other alternative. Within each species, either none of the paths are supported by the data (gray), or only one path is supported (light orange), or both paths are supported (orange and dark orange). As data, we consider the gene annotations from Ensembl and the RNA-seq evidence compiled from public databases. When both paths are supported, we highlight the cases in which they are differentially expressed in at least one tissue in dark orange. The white cells indicate species in which a one-to-one ortholog of the human query gene could not be found. (B) For each species, the percentages of events supported by both gene annotations and RNA-seq (in green), by only RNA-seq (in yellow), by only gene annotations (in blue), or unsupported (in gray) are reported. An event is considered to be supported only if both its canonical and alternative subpaths are detected.

Evolutionary splicing graphs for AS conservation

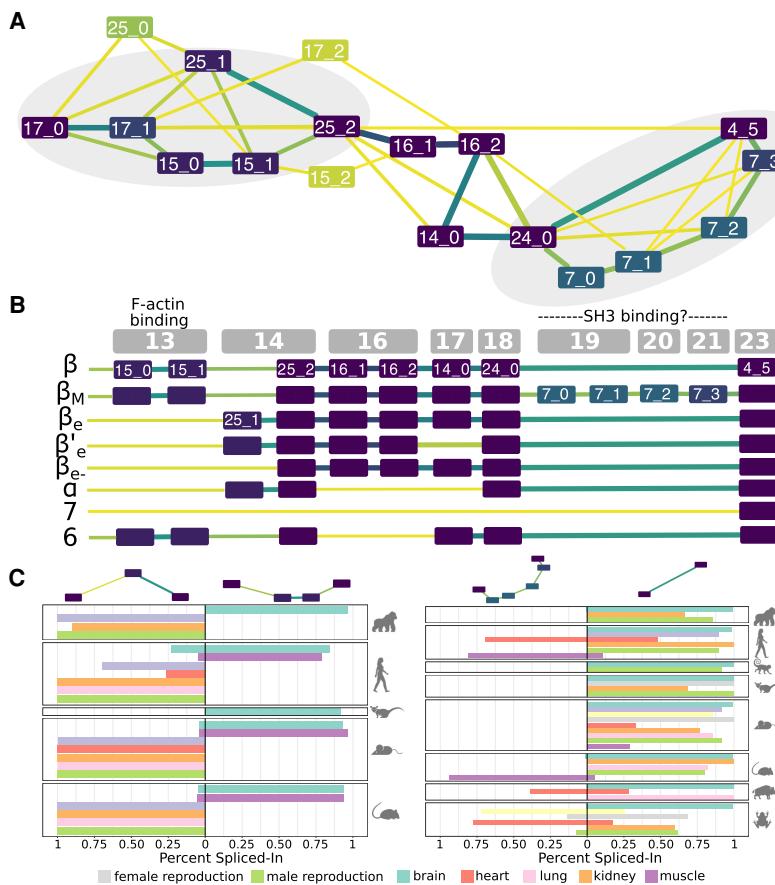


Figure 3. Transcript variability in the CAMK2B linker. (A) Evolutionary splicing subgraph computed by ThorAxe starting from 63 transcripts annotated in 10 species. It corresponds to the region linking the kinase and hub domains of CAMK2B. The colors of the nodes and the edges indicate conservation levels, from yellow (low) to dark purple (high). Conservation is measured as the *species fraction* for the nodes (proportion of species where the s-exon is present) and as the *averaged transcript fraction* for the edges (averaged transcript inclusion rate of the s-exon junction). For ease of visualization, we filtered out the s-exons present in only one species. The events documented in the literature are located in the gray areas. (B) On top, genomic structure of the human gene. Each gray box corresponds to a genomic exon (nomenclature taken from Sloutsky and Stratton 2020). (Below) List of human transcripts. All of them have been described in the literature, referred to as β (Bulleit et al. 1988), β_M (Bayer et al. 1998), β_e (Brocke et al. 1995), β'_e (Brocke et al. 1995), $\beta_{e'}$ (Cook et al. 2018), α (Bulleit et al. 1988), 7 (Wang et al. 2000), and 6 (Wang et al. 2000). The functional roles of some exons (Bayer et al. 1998; Khan et al. 2019) are given. (C) Percent-spliced in (PSI) computed from RNA-seq splice junctions for the two documented AS events. The two pairs of alternative subpaths depicted on top are also highlighted on A.

when removing the two best-annotated species, namely, human and mouse (Supplemental Fig. S5A), and when scaling up to about 100 species (Supplemental Fig. S5B). Furthermore, RNA-seq mapping revealed evolutionarily conserved tissue regulation for both events (Fig. 3C). For instance, the alternatively spliced F-actin binding region comprised of the s-exons 15_0 and 15_1 is specifically expressed in the brain and muscles of primates and rodents (Fig. 3C, on the left).

ThorAxe summarizes within- and across-species variations at the human proteome scale

We further assessed ThorAxe on the whole human proteome (18,226 genes) (Supplemental Methods). ThorAxe analysis across 12 species completed in less than 20 h with 15 cores. The genes are well-represented in all mammals (Supplemental

Fig. S6), except in platypus, which covers only 40% of the human protein-coding genome. Frog, zebrafish, and nematode cover about 65%, 50%, and 14% of the genes, respectively (Supplemental Fig. S6). ThorAxe produced ESGs with 26 s-exons, on average, and at most 354 (Supplemental Table S4). They are either very lowly or very highly conserved, as measured by the *species fraction* (SF) that is the proportion of species where a s-exon is found (Supplemental Figs. S7, S8).

We distinguish the species-specific s-exons detected in only one species and thus containing only one sequence in their MSA, from the s-exons conserved in at least two species. The proportion of species-specific s-exons goes from <10% in mammals and zebrafish to 23% in frog and 72% in nematode (Fig. 4A; Supplemental Table S5), emphasizing the high sequence divergence of this organism. These s-exons are often located at the transcript extremities (Fig. 5A,B, nodes in yellow) and tend to be smaller than the conserved ones (Supplemental Fig. S9). The vast majority of the latter are well-conserved from primates to amphibians, and their species representativity strongly correlates with the evolutionary distance they span (Fig. 4A). For instance, almost all the conserved s-exons present in frog also comprise sequences coming from primates, non-primate eutherians, and noneutherian mammals (Fig. 4A, pink curve). This correlation is even more evident in the subset of 13,558 genes with one-to-one orthologs in more than seven species (Supplemental Figs. S10, S11). Overall, ~40% of the conserved s-exons present in human span an evolutionary time of more than 400 million years, up to zebrafish (Fig. 4A). The proportion drops down to <10% outside of vertebrates.

Nevertheless, we identified 295 genes for which ThorAxe assigned most of the exonic sequences contributed by nematode to well-conserved s-exons. This set is enriched in genes coding for proteins involved in the transcription or the translation (RNA polymerases, ribosome, spliceosome, chaperones) or in protein degradation (proteasome).

As for the s-exon usage, almost a third is involved in some event (Fig. 4B). The most (respectively, least) conserved ones, are involved in deletions (respectively, insertions) (green and pink curves). This observation can be explained by the fact that ThorAxe detects events as variations from a reference canonical transcript chosen for its high conservation and length (Supplemental Methods). The alternatively expressed s-exons located at the beginning or in the middle of the protein (gold and light blue curves) tend to be more conserved than at the end of the protein (dark blue curve).

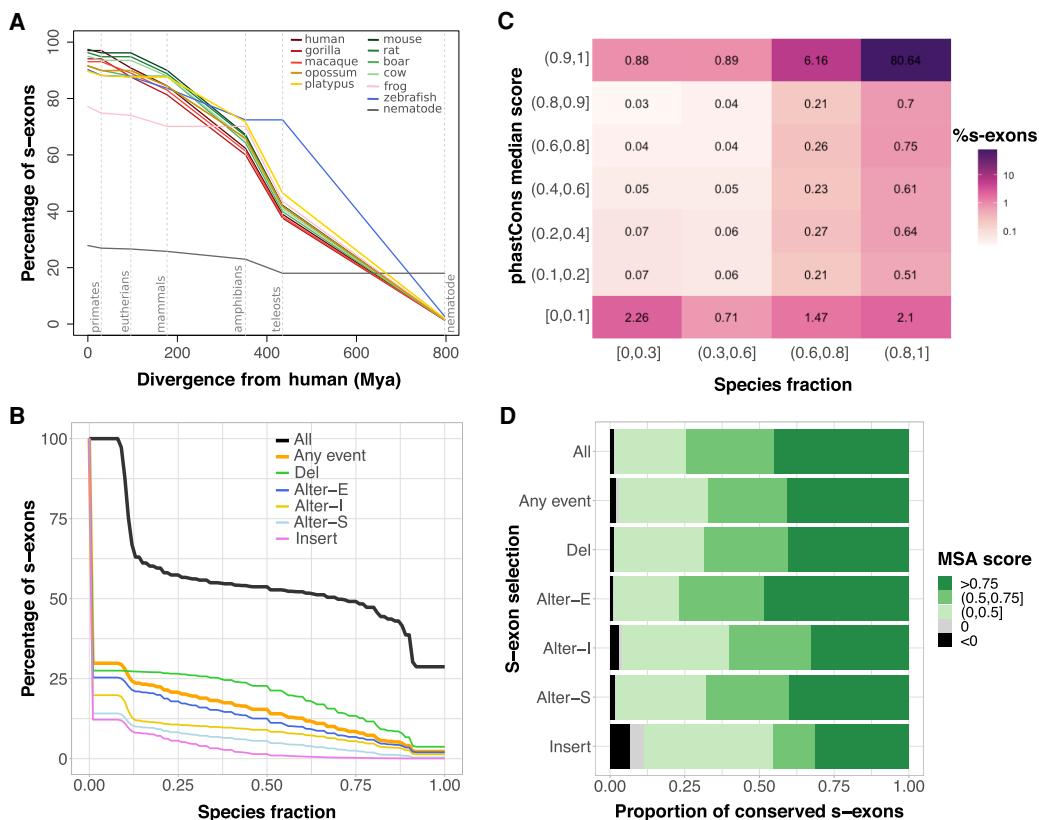


Figure 4. S-exon evolutionary profiling over the whole protein-coding human genome. (A) Percentages of s-exons conserved at different evolutionary distances from human (represented by dashed vertical lines). Each curve is centered on its corresponding species. The values at the origin are the percentages of conserved (i.e., not species-specific) s-exons. Conservation is then assessed at each evolutionary distance according to the s-exons possessing at least one representative in each phylogroup. For instance, we report 73%–76% of the s-exons of frog (pink curve) as conserved among eutherians (second dashed line) in the sense that they are also conserved in at least one primate (among human, gorilla, macaque) and at least one nonprimate eutherian (among rat, mouse, boar, cow). Likely, conservation up to mammals (68%–72% for frog) would imply at least one primate, one nonprimate eutherian, and one noneutherian mammal. See also Supplemental Figure S10 for a version of this plot focusing on genes with one-to-one orthologs in more than seven species. (B) Cumulative distributions of s-exon species fraction. On the y-axis we report the percentage of s-exons with a species fraction greater than the x-axis value. The different curves correspond to all s-exons (All), only those involved in at least an event (Any event), or only those involved in a specific type of event. (Alter-S) alternative start; (Alter-I) alternative (internal); (Alter-E) alternative end; (Del) deletion; (Insert) insertion. (C) Heatmap of the s-exon phastCons median scores versus the s-exon species fractions. Only the s-exons longer than 10 residues and belonging to genes with one-to-one orthologs in more than seven species are shown. (D) Proportions of conserved s-exons displaying very poor (negative score) to very good (score close to one) alignment quality. The MSA score of a s-exon is computed as a normalized sum of pairs. A score of 1 indicates 100% sequence identity without any gap. The proportions are given for different s-exon selections (same labels as in B).

The s-exons accurately measure sequence conservation

To evaluate ThorAxe ability to correctly match exonic sequences between more or less distant species, we compared the s-exon species fractions with estimates of evolutionary conservation deduced from whole-genome alignments between human and 99 other vertebrates, available as phastCons scores through the UCSC Genome Browser (Supplemental Methods; Siepel and Haussler 2005; Siepel et al. 2005). Overall, the two measures agree very well (Fig. 4C). For instance, most of ThorAxe species-specific or very lowly conserved s-exons ($SF < 0.3$) are not expected to be evolutionarily conserved based on genomic alignments (Fig. 4C, left column). Nevertheless, ThorAxe seems to underestimate the conservation level of 1508 s-exons (Fig. 4C, top left corner). We investigated whether these s-exons could share significant sequence similarity with some other s-exons defined across distinct species, and we found that only 24 of them may be considered as “false negatives” (Supplemental Table S6). Hence, the low conservation

estimated by ThorAxe likely reflects the lack of annotated transcripts in certain species rather than errors in the heuristic. Reciprocally, ThorAxe seems to detect more conservation signal than whole-genome alignments for more than 7000 s-exons ($SF > 0.3$) (Fig. 4C, bottom row), without any particular trend in their alternative usage (Supplemental Fig. S12). They display high sequence identity (Supplemental Fig. S12C,D), suggesting that they are indeed conserved across many species and not “false positives.” The collagen type XVIII alpha 1 chain (*COL18A1*) on its own contributes 12 such s-exons, conserved from primates to amphibians according to ThorAxe ($SF > 0.8$) but with very low phastCons scores (< 0.1). The *COL18A1* protein is highly enriched in glycines and prolines and the 12 s-exons fall within regions of low sequence complexity (Supplemental Fig. S13). We can hypothesize that this low-complexity context confounds the whole-genome alignments but not ThorAxe heuristic. To get a better view on the s-exon sequence divergence, we computed the sum-of-pair scores of the associated MSAs (Supplemental

Evolutionary splicing graphs for AS conservation

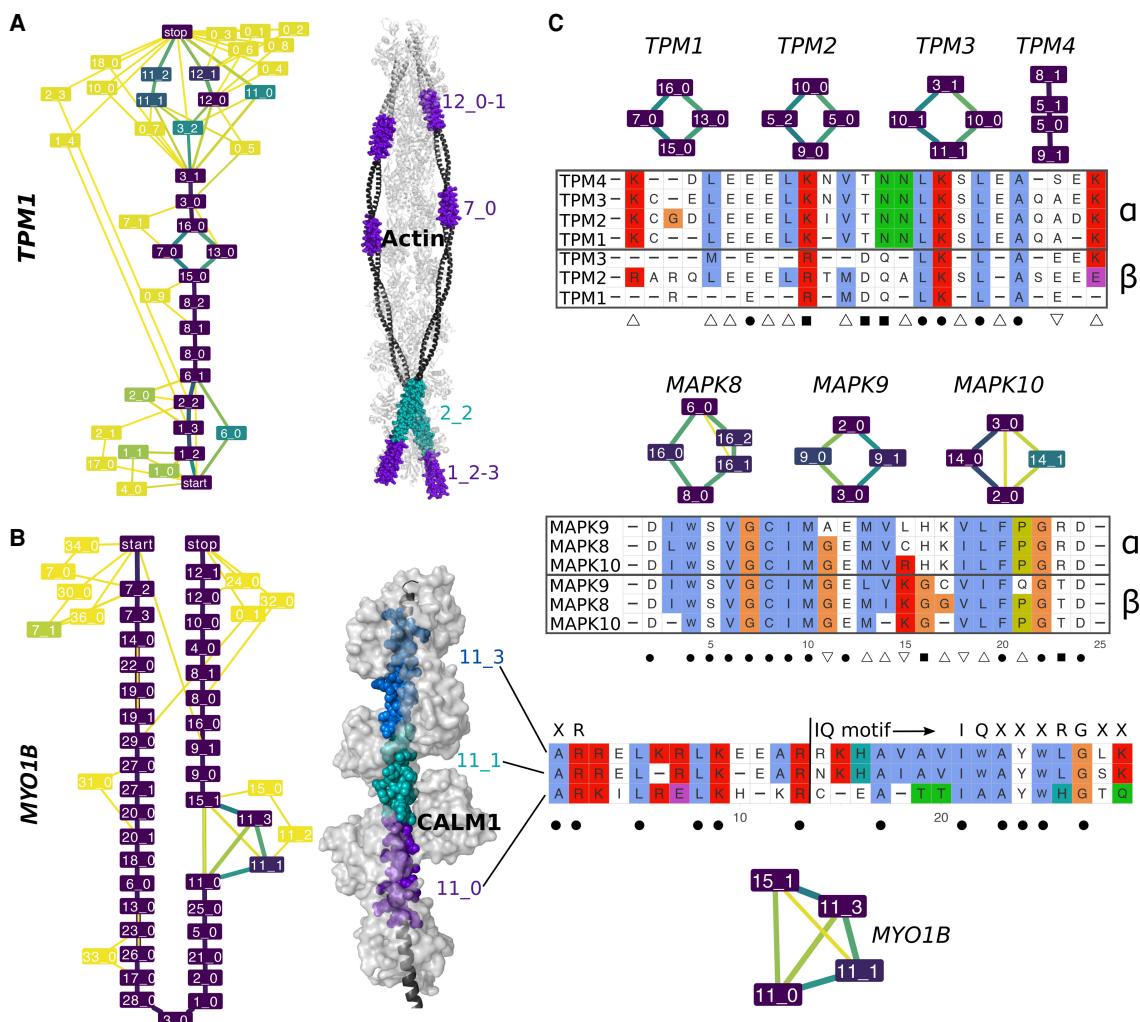


Figure 5. Examples of evolutionarily conserved events with in-gene paralogy. (A,B) ESGs computed by ThorAxe (left) and the best 3D templates found by HHblits (right); PDB codes 2w49:abuv (Wu et al. 2010) and 2dfs:H (Liu et al. 2006) for TPM1 and MYO1B. On the ESGs, the colors indicate conservation levels, species fraction for the nodes and averaged transcript fraction for the edges (Supplemental Methods). The nodes in yellow are species-specific, whereas those in dark purple are present in all species. The 3D structures show complexes between the query proteins (black) and several copies of their partners (light gray). The s-exons involved in conserved events are highlighted with colored spheres. (C) S-exon consensus sequence alignments within a gene family (TPM on top, MAPK in the middle) or a gene (MYO1B, at the bottom). Each letter reported is the amino acid conserved in all sequences of the corresponding MSA (allowing one substitution). The color scheme is that of Clustal X (Thompson et al. 1997). The subgraphs show the events in which the s-exons are involved. The symbols α and β on the right indicate groups of s-exons defined across paralogous genes based on sequence similarity (Supplemental Methods). The symbols at the bottom denote highly conserved positions across the gene family: (dot) fully conserved position; (square) position conserved only within each s-exon group; (upward triangle) position conserved in the α group only; (downward triangle) position conserved in the β group only. For MYO1B, the start and canonical sequence of the CALM1-binding IQ motif are indicated. The motifs resulting from different combinations of the depicted s-exons are numbered 4, 4/5, and 4/6 in the literature (Greenberg and Ostap 2013).

Methods). Overall, almost half of the conserved s-exons have very high-quality MSAs with very few mismatches and gaps (Fig. 4D, score > 0.75). This proportion increases up to about 70% on the 50-gene set (Supplemental Fig. S14). A very small proportion (about 1%) of s-exons have very poor quality MSAs (Fig. 4D, in black), and those are typically short (Supplemental Fig. S15). Moreover, the inserted and, to a lesser extent, alternatively expressed s-exons display lower-quality MSAs (Fig. 4D, Insert and Alter-I). Finally, we checked the relationship between structural order/disorder and s-exon sequence divergence. Structurally disordered s-exons (~40% of the ensemble) tend to be less conserved and to have lower-quality MSAs than well-folded ones

(Supplemental Fig. S16). However, the differences between the two groups are rather small.

The comparison of similar s-exons unveils functionally relevant signatures

ThorAxe allows exploring how function diversification may arise through the alternative usage of similar sequences within and across genes. We illustrate the power of the approach on three gene families (Fig. 5; see also Supplemental Tables S2, S3), focusing on a set of events involving two or more highly conserved s-exons with similar consensus sequences. The origin of the events can be

traced back to the ancestor common to mammals, amphibians, and fishes. The first example is given by the tropomyosin family (Fig. 5A,C), whose protein members (TPM1,2,3,4) serve as integral components of the actin filaments forming the cell cytoskeleton. Several conserved events detected in the ESGs have direct implications for actin binding (Fig. 5A; Wu et al. 2010; Pathan-Chhatbar et al. 2018). Among them, the internal mutually exclusive pair displays high sequence similarity and strong sequence conservation across species and between paralogous genes (Fig. 5C, on top, α and β groups). Fourteen specificity-determining sites (SDS) can be identified (Fig. 5C). SDS are key positions with specific conservation patterns, and they play a role in diversifying protein function in evolution (Chakraborty and Chakrabarti 2015). Given two groups, here α and β , type I SDS are conserved in one group and variable in the other one, indicating different functional constraints between the groups. For instance, position 24 is occupied by a glutamate in all the s-exons from the β group, whereas it is variable in the α group. Type II SDS are conserved in both groups, but each group displays a different amino acid. This is the case of positions 14 and 15, where the Thr-Asn couple of the α group is replaced by Asp-Gln in the β group. These SDS may be responsible for the differences observed in actin filaments formation, mobility, and myosin recruitment ability between the isoforms (Pathan-Chhatbar et al. 2018). The Mitogen-activated protein kinase (MAPK) family (MAPK8,9,10) gives another example with even higher sequence identities (Fig. 5C, in the middle). Among the eight identified SDSs, three positively charged residues—His, Lys, and Arg in positions 16, 17, and 23—are specifically conserved in the α group, whereas the β group is characterized by Lys, Gly, and Thr in positions 15, 16, and 23. These observations are in line with our previous study highlighting differences in the dynamical behavior of these residues (Ait-hamlat et al. 2020) and their potential implication for substrate selectivity (Waetzig and Herdegen 2005). As a third example, myosin IB comprises a set of consecutive similar s-exons overlapping with calmodulin (CALM1)-binding so-called IQ motifs (Fig. 5B). The alternative inclusion of two s-exons, which share almost 50% identity (Fig. 5C, bottom), results in different binding motifs. Compared to the motif's canonical form (IQXXXRGXXXR) (Houdusse et al. 1996; Bähler and Rhoads 2002), they all lack the glutamine in the IQ residue pair and the arginine in the RG pair. These differences could explain their lower affinity compared to the constitutive s-exons (Greenberg and Ostap 2013).

Alternative usage of similar sequences is not a rare phenomenon

At the human genome scale, we identified 2190 genes (12% of the protein-coding genome) with evidence of evolutionarily conserved alternative usage of similar exonic sequences (Fig. 6). The corresponding proteins tend to be involved in cell organization and muscle contraction (cytoskeleton, collagen, fibers, etc.), and in intercellular communication (Supplemental Fig. S17). Our strategy here was to look for similar s-exon pairs involved in some event (Supplemental Methods). We found a total of 31,031 pairs, among which 446 are mutually exclusive (Fig. 6A,B, MEX). This case scenario highlights the exclusive usage of one or the other version of a protein region. The 232 concerned proteins are enriched in transporters and channels (Supplemental Fig. S17). Another 438 pairs (coming from 134 genes) are alternatively used without mutual exclusivity (Fig. 6A,B, ALT). In about half of the MEX or ALT s-exon pairs, one of the s-exons is conserved in all studied species, and the other one in more than half of them (Fig. 6C). In 3813

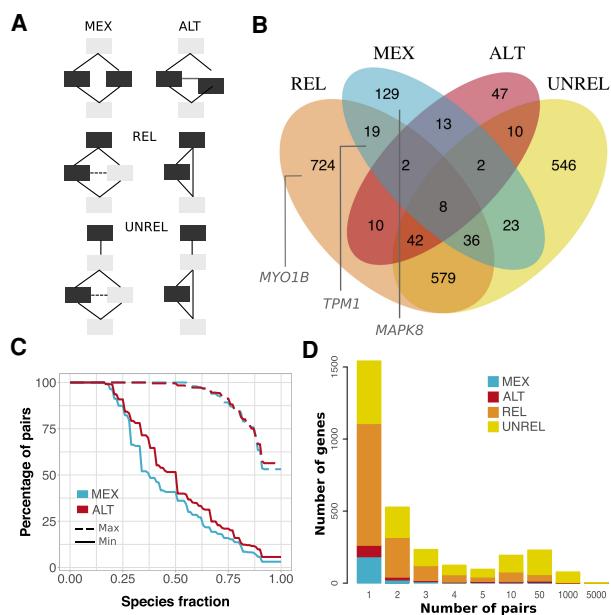


Figure 6. Alternative usage of similar s-exons. (A) Evolutionary splicing subgraphs depicting different alternative usage scenarios. The detected s-exon pairs are colored in black. (MEX) mutual exclusivity; (ALT) alternative (non-mutually exclusive) usage; (REL) one s-exon is in the canonical or alternative subpath of an event (of any type), whereas the other one serves as a “canonical anchor” for the event; (UNREL) one s-exon is in the canonical or alternative subpath of an event (of any type), whereas the other one is located outside the event in the canonical transcript. Each detected pair is assigned to only one category with the following priority rule: MEX > ALT > REL > UNREL. (B) Venn diagram of the genes containing similar pairs of s-exons. The genes shown in Figure 5 are highlighted in the corresponding subsets. (C) Cumulative distributions of s-exon conservation. On the y-axis we report the percentage of s-exon pairs with species fraction greater than the x-axis value. The solid (respectively, dashed) curve corresponds to the highest (respectively, lowest) species fraction among the two s-exons in the pair. We report values only for the MEX (blue) and ALT (red) categories. (D) Distribution of per-gene s-exon pair number within each of the four categories. For instance, the yellow rectangle at $x=50$ gives the number of genes with more than 10 and up to 50 UNREL s-exon pairs.

pairs, one s-exon is included in the canonical or alternative subpath of an event, whereas the other one serves as a “canonical anchor” for the event (Fig. 6A,B, REL). This highlights the AS-induced modulation of the number of nonidentical consecutive copies of a protein region. The remaining 26,334 pairs correspond to cases in which one of the s-exons participates in an event (on the canonical or alternative subpath), whereas the other one is located outside the event in the canonical transcript (UNREL). The full lists of s-exons are given in Supplemental Tables S7–S9. This resource overlaps well with a previously reported manually curated set of 97 human genes with mutually exclusive homologous exon pairs (Abascal et al. 2015). It extends it by one order of magnitude and represents a more diverse range of AS-mediated relationships between similar protein regions. Although most of the identified genes contain only one or a few similar s-exons pair(s), almost 50 genes have several hundreds or thousands of pairs (Fig. 6D). Nebulin gives the most extreme example, with 2380 detected pairs. This giant skeletal muscle protein has evolved through several duplications of nebulin domains, and a definition of pertinent nebulin evolutionary units was proposed (Björklund et al. 2010). These units correspond to parts of exons, in line with ThorAxe s-exons MSAs.

Evolutionary splicing graphs for AS conservation

Comparison with other studies

We evaluated the ability of ThorAxe to detect the events reported in two reference studies dealing with the evolution of AS (Barbosa-Morais et al. 2012; Merkin et al. 2012). We could map 40 exons from Barbosa-Morais et al. (2012) and 323 exons from Merkin et al. (2012), all displaying conserved tissue-specific splicing patterns, onto our ESGs (Supplemental Methods). For >75% of the 40 exons, we found events conserved across mammals and ranked first or second in the ESG (Supplemental Fig. S18A, dark green). The conservation signal extends to amphibians for 18 events and to teleosts for five. For the 323-exon set, we detected 277 events, 90% of which are in the top three most conserved of ThorAxe ESGs. About 70% are well-conserved in mammals, 82 in amphibians, and 19 in teleosts (Supplemental Fig. S18B). In particular, we can mention exon 3 from the eukaryotic translation elongation factor 1 delta (*EEF1D-ex3*) and exon 20 from the tight junction protein 1 (*TJP1-ex20*) highlighted in Figures 2 and 4 from Merkin et al. (2012). In both cases, the deletion of the matching s-exon(s) is the most conserved event of the ESG and is observed in human, mouse, and pig. The deletion of *EEF1D-ex3* (s-exons 1_1 and 1_2) is also conserved in gorilla and cow, whereas that of *TJP1-ex20* (s-exon 6_0) is also conserved in macaque. We can also pinpoint the six exons intersecting with our curated set (Supplemental Fig. S19). ThorAxe detected events well-conserved across mammals for all of them (and in amphibians, for two of them), with RNA-seq evidence of conserved tissue-specific AS patterns for all but one. One of the matching s-exons, 11_1 from *MYO1B*, is part of a couple of alternatively spliced pseudorepeats (Fig. 5B). Although we found conserved tissue-regulation patterns for 11_3, the other s-exon in the couple, it was not reported in Merkin et al. (2012). This example highlights the difficulty of assessing the tissue-specific expression of several instances of (pseudo-)repeated sequences, and showcases ThorAxe's ability to deal with such complexity.

Comparison with other methods

We assessed the pertinence of the ThorAxe heuristic by performing an ablation study and by comparing it with two popular exon orthology detection methods. For the ablation study, one strategy was to skip the exon clustering step (Methods, step b), and the other was to rely solely on global multiple sequence alignment, which means both the exon clustering step and the s-exon refinement step are skipped (Methods, steps b and e). Compared to these two strategies, ThorAxe produces longer and higher-quality s-exon MSAs (Supplemental Fig. S18C,D). Specifically, the clustering step helps to improve the s-exon sequence identities (Supplemental Fig. S18D, cf. blue and orange boxes) by reducing the space of sequences to align. The final local optimization step increases the lengths of the s-exons (Supplemental Fig. S18C, cf. blue and red boxes) by minimizing subexon boundaries violations. As popular exon orthology detection methods, we chose the Reciprocal Best Blast Hit (RBBH) method and Ensembl Compara (Herrero et al. 2016). The RBBH method consists in finding the best matching subexon pairs across any two species using BLAST. One of the drawbacks of this method is that many subexons remain without any match in other species (Supplemental Fig. S18E, orange box). By allowing for one-to-many subexon matching between species, ThorAxe covers a much higher proportion of subexons (Supplemental Fig. S18E, blue box). ThorAxe strategy is justified by the fact that exons may undergo truncation or elongation in the course of evolution, and thus we do not expect a one-to-one

relationship between them across a pair of species. Moreover, ThorAxe increased subexon coverage is not at the expense of sequence identity (Supplemental Fig. S20). Another drawback of RBBH is that defining s-exons from a set of pairwise alignments of subexons is a difficult task. Finally, Ensembl Compara relies on whole-genome alignments. However, it does not include all the species for which annotated transcripts are available in Ensembl. Moreover, one can expect that working with DNA sequences produces lower-quality alignments compared to working with protein sequences, as is done by ThorAxe.

Assessment of the default parameter choices

All parameters in ThorAxe are customizable by the user, enabling a rapid adaptation of the method to specific contexts and questions. We investigated the pertinence of some of the default values. For instance, by default, ThorAxe filters out the transcripts flagged in Ensembl as lowly supported (Transcript Support Level [*TSL*] < 3). This restriction only concerns human and mouse, because the other species do not have any TSL annotations. By varying the TSL value between 1 and 5, we observed that, as expected, the less stringent the TSL criterion, the higher the number of transcripts and of events (Supplemental Fig. S21A, B). However, very little change is observed in the definition of the canonical transcript and in the conservation levels of the s-exons, suggesting that the results and their interpretation are robust to this parameter (Supplemental Fig. S21C–E). Globally, the biggest changes are observed either when only top-quality transcripts are retained (*TSL* ≤ 1) or when transcripts are not filtered at all (*TSL* ≤ 5). We thus recommend using intermediate TSL values (2–4).

The first step of the ThorAxe algorithm consists in grouping similar exonic sequences together, with the aim of reducing the complexity of the subsequent construction of the MSAs. By default, ThorAxe applies a sequence identity cutoff of 30% to define the clusters. To assess the suitability of this cutoff value for handling divergent sequences, we looked at the MSA quality with respect to the species fraction (Supplemental Fig. S22). Although the two measures are correlated, a significant portion of s-exons display high species fractions (>0.8) but low MSA scores (≤0.5). This observation suggests that ThorAxe is able to cluster together divergent sequences that are difficult to align. The cutoff may be adapted by the user depending on the level of sequence divergence expected in the input data.

To ease interpretability of the results and ensure that the s-exons represent groups of one-to-one orthologous exonic regions, ThorAxe default mode considers only one-to-one orthologous genes, as annotated in Ensembl. This means that organisms with additional round(s) of whole-genome duplications and/or separated by long evolutionary distances will likely be excluded from the analysis. We found that taking into account many-to-many gene orthology relationships leads to a better detection of the documented events (Supplemental Fig. S23). The improvement is particularly visible for zebrafish, in which we now have eight events with both the canonical and alternative subpaths supported by Ensembl annotations (Supplemental Fig. S23, cf. A and B). The detection in rat, cow, and platypus is also improved. Finally, we tested the impact of excluding the two best-annotated species, namely, human and mouse. As a result of this exclusion, three documented events are lost. Nevertheless, the conservation profiles of the other events remain almost identical (Supplemental Fig. S23, cf. A and C).

Discussion

We have presented a novel method to describe transcript variability in evolution. Our approach provides a double generalization, by extending the definition of SG to the case of multiple species and by providing a way to combine MSAs over structures with a partial order. The heuristic is general enough to deal with very different genes (in terms of length, structure, degree of conservation, number of transcripts, etc.). Its identification of transcript minimal building blocks (the s-exons) is the first and necessary step for inferring evolutionary scenarios explaining AS-induced protein function diversification (Ait-hamlat et al. 2020). Our data structure is reminiscent of current developments on pangenome graphs. However, pangenome approaches keep track of variations across a population, whereas we highlight conservation across species in the context of AS. To the best of our knowledge, we are the first to do it. Effectively, we consider a *pan-transcriptome* across multiple species. As a consequence, we do not need to rely on a central species and project the transcripts on it. In the analysis conducted here, human was taken as a reference only to find orthologous genes in other species.

To illustrate the potential of the method, we assessed the evolutionary conservation and tissue regulation of a set of documented AS events we compiled from the literature. This set could serve as a reference for future studies. We then scaled up to the human protein-coding genome, and found that AS is conserved across a wide range of evolutionary distances, is not limited to ancient events, and does not generate conserved alternative isoforms in all of the proteins. We have shown that the alternative usage of repeats in protein is not a rare phenomenon in the human proteome and that it is of ancient evolutionary origin. Although we focused on one-to-one orthologs, thereby limiting the contribution of nematode, our analysis can be readily extended to one-to-many orthology relationships to better compare vertebrates with other organisms.

On the one hand, a limitation of the approach is that it mainly relies on gene annotations, which may be partial, incomplete, or erroneous (Salzberg 2019). To avoid errors, we chose to select only high-quality transcripts, with the risk of biasing the results toward species with more fully annotated alternative splicing landscapes. Another strategy could be to use APPRIS annotations, but we expect a reduction in the overall input transcript variability, therefore limiting ThorAxe potential to discover AS events. Moreover, APPRIS annotations are derived from an analysis accounting for transcript sequence conservation, which would be somewhat redundant with ThorAxe's own analysis of AS conservation. On the other hand, an important advantage of ThorAxe is its robustness with respect to the presence of highly divergent sequences and the creation of species-specific s-exons. Indeed, the latter simply contribute single nodes to the ESG without preventing the detection of conserved AS events. In a way, detecting too many species-specific s-exons would not be a problem because this would only slightly diminish ThorAxe conservation estimates without hampering the interpretation of the ESGs. Traditional methods may recover genomic conservation at lower levels of sequence similarity, but by disregarding the whole transcript structure, they may not properly evaluate AS conservation.

Future work could benefit from the development of more accurate approximations of the general problem stated here. Another direction is to expand the application field to transcriptomes coming from patients or human populations. In the coming years, we expect a tremendous increase in available transcriptomic data,

including transcriptome annotations generated by long-read sequencing technologies (Byrne et al. 2019). Methods addressing the complexity of these data will become instrumental in understanding the evolution of a disease, for example, cancer, and the phenotypic variability among human populations and individuals (Lonsdale et al. 2013; Park et al. 2018). ThorAxe could be easily adapted to deal with these data, and, along this line, we have already implemented the possibility to give additional "user-defined" transcripts as input.

Methods

Complexity of the problem: determining a minimal ESG is NP-hard

To illustrate the complexity of determining a minimal ESG, consider a case example with n input transcripts observed in n species (i.e., one transcript per species). Moreover, because the problem is theoretically independent of the penalties σ_l and σ_s (Equation 1), an algorithm that would solve it in the general case would also be valid for $\sigma_l = \sigma_s = +\infty$. In this scenario, a minimal ESG has no edge and maximizes the sum-of-pair-score σ . Thus, the problem of building a minimal ESG is equivalent to solving the problem of multiple sequence alignment with sum-of-pair-score σ on the n input transcripts. Because the n input transcripts can be any string (over the amino acid alphabet), and finding a MSA of any string with sum-of-pair-score is NP-hard (Wang and Jiang 1994), it follows that finding a minimal ESG is NP-hard.

Description of ThorAxe algorithm and parameters

Given a gene name and a list of species as input, ThorAxe extracts and exploits gene annotations from Ensembl (and, optionally, input transcripts provided by the user) to build an ESG maximizing the sequence similarity within each node (or s-exon) and minimizing the number of induced edges, which indirectly implies that the number of nodes is minimized. The heuristic approximates the best-scored solution of Equation 1 by controlling the creation of induced edges, without explicitly computing ESG scores. It unfolds in six main steps (Fig. 1C; Supplemental Fig. S1).

- Data acquisition and preprocessing. ThorAxe downloads the gene tree, the transcripts annotated as protein coding and their exons (genomic coordinates, sequences, and phases) for the query gene and its (by default one-to-one) orthologs in the selected species from Ensembl. ThorAxe then removes incomplete or lowly supported ($TSL < 3$, adjustable by the user) transcripts, and translates the retained transcripts' DNA sequences into amino acid sequences using the exon phases. Transcripts or exonic regions leading to the same protein sequence are merged, but the same genomic region may lead to the generation of several protein sequences if it is associated with more than one frame (Supplemental Methods). ThorAxe can additionally take as input user-defined transcripts (from any species). The format is similar to the one in Ensembl and includes exon coordinates, their rank, frame, and nucleotide sequence.
- Pairwise-alignment-based exon preclustering. ThorAxe clusters the input exons based on their sequence similarity (Fig. 1C, three clusters colored in red, cyan, and blue). This step provides a coarse-grained partitioning of the sequence space that reduces the complexity of step d (see below). We perform pairwise local alignments using a modified version of the Hobohm I algorithm (Supplemental Methods). We use a relatively low default sequence identity threshold of 30% to ensure

Evolutionary splicing graphs for AS conservation

- homology detection across many species. As illustrated in Figure 1C by cluster 2, pairs of duplicated mutually exclusive exons coming from the same species will likely be grouped in the same cluster (see also *Supplemental Fig. S1*).
- c. Redundancy reduction. ThorAxe defines a set of subexons for each species. This implies systematically detecting overlapping exons and replacing them by nonredundant distinct subexons. In the example shown in Figure 1C, one exon from gorilla leads to the definition of two subexons (in red; for their sequences, see also *Supplemental Fig. S2A*). This step relies only on the genomic coordinates of the exons and does not require aligning the exonic sequences. It is performed after exon clustering, because dealing with subexons at this early stage would add some unnecessary complexity by augmenting the number of comparisons and the ambiguity associated with small sequences.
 - d. MSA-based s-exon identification. ThorAxe defines a set of s-exons across all species, by aligning the exonic sequences belonging to each cluster defined in step b and identifying blocks in the constructed MSAs (Fig. 1C, see the three MSAs corresponding to the three clusters). The aim of this step is to determine a mapping of exonic sequences between different species (for details, see *Supplemental Methods*). To identify the s-exons from each MSA, ThorAxe scans the MSA from left to right and creates a new s-exon whenever there is a change of sub-exon in at least one sequence/species (Algorithm 1 in *Supplemental Methods*). This ensures that the identified s-exons can be used as building blocks to reconstruct any transcript in any species from the input data.
 - e. S-exon refinement. ThorAxe refines the s-exons' definition by locally optimizing the MSAs built in step d. The aim is twofold, namely, to improve the quality of the MSAs associated to the s-exons and to reduce the number of s-exons, and hence the number of induced edges in the corresponding ESG. This step then represents a means to increase the ESG score expressed in Equation 1 without explicitly computing it. Specifically, we systematically detect lowly scored subexons and migrate them from one MSA to another, and we minimize the number of very small s-exons, comprising only one or two columns (*Supplemental Methods*).
 - f. ESG construction and annotation, and event detection. Once the s-exons have been identified, building the corresponding ESG is straightforward (Fig. 1C). ThorAxe annotates the nodes and the edges of the output graph with evolutionary information and summary statistics (*Supplemental Fig. S2D*; *Supplemental Methods*). Finally, it defines a canonical transcript and detects a set of events as variations between this reference transcript and each input transcript. Ideally, the canonical transcript should be well-represented across species; thus, to choose it, we rely on a combination of conservation measures computed over the ESG edges (*Supplemental Methods*; *Supplemental Fig. S24*). By default, the events are detected on a reduced version of the ESG, where the edges supported by only one transcript have been removed (Algorithm 2 in *Supplemental Methods*). We visualized the ESGs with Cytoscape V.3.7.2 (Shannon et al. 2003).

An additional output of ThorAxe is the list of input transcripts described as collections of s-exons (where each s-exon is designated by a symbol) and the gene tree representative of the selected species. These data can directly serve as input for PhyloSofS (Ait-hamlat et al. 2020), toward the reconstruction of transcripts' phylogenetic forests. ThorAxe may also be easily interfaced with other tools requiring the same type of input.

Analysis of ThorAxe results

We give details about the calculation of the MSA scores, the detection of similar pairs of s-exons, the complementation of the ESGs with RNA-seq splice junctions, the characterization of isoform 3D structures and disorder content, the functional analysis of some genes, the comparison with phastCons scores, other studies, and other methods in the *Supplemental Methods*, *Supplemental Figs. S25–S27*, and *Supplemental Table S10*.

Software availability

ThorAxe is freely available at GitHub (<https://github.com/PhyloSofS-Team/thoraxe>) and as a stand-alone package and Python module as *Supplemental Code*. All data supporting the findings of this study are available via a supplementary web-server (<http://www.lcqb.upmc.fr/ThorAxe>) and as *Supplemental Material*.

Competing interest statement

The authors declare no competing interests.

Acknowledgments

A grant of the French National Research Agency (MASSIV project, ANR-17-CE12-0009) provided a salary to D.J.Z. and funded the work of S.L. We thank P. Charpentier and J. Cortes for their help in the systematic detection of the disordered regions and S. Grudinin for insightful comments. We thank F. Oteri and H. Ripocche for the technical support. We thank the reviewers for their comments, which greatly improved the manuscript.

Author contributions: D.J.Z., H.R., and E.L. designed the research. D.J.Z. and S.L. performed the implementation. D.J.Z., S.L., H.R., and E.L. produced and analyzed the results. A.B. contributed the proof of NP-hard complexity. E.L. wrote the manuscript with support and feedback from all authors. H.R. and E.L. supervised the project.

References

- Abascal F, Tress ML, Valencia A. 2015. The evolutionary fate of alternatively spliced homologous exons after gene duplication. *Genome Biol Evol* **7**: 1392–1403. doi:10.1093/gbe/evv076
- Agosto LM, Gazzara MR, Radens CM, Sidoli S, Baeza J, Garcia BA, Lynch KW. 2019. Deep profiling and custom databases improve detection of protein forms generated by alternative splicing. *Genome Res* **29**: 2046–2055. doi:10.1101/gr.248435.119
- Ait-hamlat A, Zea DJ, Labeeuw A, Polit L, Richard H, Laine E. 2020. Transcripts' evolutionary history and structural dynamics give mechanistic insights into the functional diversity of the JNK family. *J Mol Biol* **432**: 2121–2140. doi:10.1016/j.jmb.2020.01.032
- Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ. 1990. Basic local alignment search tool. *J Mol Biol* **215**: 403–410. doi:10.1016/S0022-2836(05)80360-2
- Bähler M, Rhoads A. 2002. Calmodulin signaling via the IQ motif. *FEBS Lett* **513**: 107–113. doi:10.1016/S0014-5793(01)03239-2
- Baralle FE, Giudice J. 2017. Alternative splicing as a regulator of development and tissue identity. *Nat Rev Mol Cell Biol* **18**: 437–451. doi:10.1038/nrm.2017.27
- Barbosa-Morais NL, Irimia M, Pan Q, Xiong HY, Gueroussov S, Lee LJ, Slobodeniuc V, Kutter C, Watt S, Colak R, et al. 2012. The evolutionary landscape of alternative splicing in vertebrate species. *Science* **338**: 1587–1593. doi:10.1126/science.1230612
- Bayer KU, Harbers K, Schulman H. 1998. α KAP is an anchoring protein for a novel CaM kinase II isoform in skeletal muscle. *EMBO J* **17**: 5598–5605. doi:10.1093/emboj/17.19.5598
- Bhattacharya M, Lee YK, Muratcioglu S, Qiu B, Nyayapati P, Schulman H, Groves JT, Kuriyan J. 2020. Flexible linkers in CaMKII control the balance between activating and inhibitory autophosphorylation. *eLife* **9**: e53670. doi:10.7554/eLife.53670

- Birzele F, Csaba G, Zimmer R. 2008. Alternative splicing and protein structure evolution. *Nucleic Acids Res* **36**: 550–558. doi:10.1093/nar/gkm1054
- Björklund AK, Light S, Sagit R, Elofsson A. 2010. Nebulin: a study of protein repeat evolution. *J Mol Biol* **402**: 38–51. doi:10.1016/j.jmb.2010.07.011
- Brocke L, Srinivasan M, Schulman H. 1995. Developmental and regional expression of multifunctional Ca²⁺/calmodulin-dependent protein kinase isoforms in rat brain. *J Neurosci* **15**: 6797–6808. doi:10.1523/JNEUROSCI.15-10-06797.1995
- Bullett RF, Bennett MK, Molloy SS, Hurley JB, Kennedy MB. 1988. Conserved and variable regions in the subunits of brain type II Ca²⁺/calmodulin-dependent protein kinase. *Neuron* **1**: 63–72. doi:10.1016/0896-6273(88)90210-3
- Byrne A, Cole C, Volden R, Vollmers C. 2019. Realizing the potential of full-length transcriptome sequencing. *Philos Trans R Soc Lond B Biol Sci* **374**: 20190097. doi:10.1098/rstb.2019.0097
- Chakraborty A, Chakrabarti S. 2015. A survey on prediction of specificity-determining sites in proteins. *Brief Bioinform* **16**: 71–88. doi:10.1093/bib/bbt092
- Christinat Y, Moret BM. 2012. Inferring transcript phylogenies. *BMC Bioinformatics* **13** (Suppl 9): S1. doi:10.1186/1471-2105-13-s9-s1
- Climente-González H, Porta-Pardo E, Godzik A, Eyras E. 2017. The functional impact of alternative splicing in cancer. *Cell Rep* **20**: 2215–2226. doi:10.1016/j.celrep.2017.08.012
- Cook SG, Bourke AM, O'Leary H, Zaegel V, Lasda E, Mize-Berge J, Quillinan N, Tucker CL, Coultrap SJ, Herson PS, et al. 2018. Analysis of the CaMKIIα and β splice-variant distribution among brain regions reveals isoform-specific differences in holoenzyme formation. *Sci Rep* **8**: 5448. doi:10.1038/s41598-018-23779-4
- de la Fuente L, Arzalluz-Luque A, Tardáguila M, del Risco H, Martí C, Tarazona S, Salguero P, Scott R, Lerma A, Alastrue-Agudo A, et al. 2020. tappAS: a comprehensive computational framework for the analysis of the functional impact of differential splicing. *Genome Biol* **21**: 199. doi:10.1186/s13059-020-02028-w
- De La Grange P, Grataud L, Delord M, Dutertre M, Auboeuf D. 2010. Splicing factor and exon profiling across human tissues. *Nucleic Acids Res* **38**: 2825–2838. doi:10.1093/nar/gkq008
- Denti L, Rizzi R, Beretta S, Della Vedova G, Previtali M, Bonizzoni P. 2018. ASGAL: aligning RNA-Seq data to a splicing graph to detect novel alternative splicing events. *BMC Bioinformatics* **19**: 444. doi:10.1186/s12859-018-2436-3
- Ezkurdia I, Rodriguez JM, Carrillo-de Santa Pau E, Vázquez J, Valencia A, Tress ML. 2015. Most highly expressed protein-coding genes have a single dominant isoform. *J. Proteome Res* **14**: 1880–1887. doi:10.1021/pr501286b
- González-Porta M, Frankish A, Rung J, Harrow J, Brazma A. 2013. Transcriptome analysis of human tissues and cell lines reveals one dominant transcript per gene. *Genome Biol* **14**: R70. doi:10.1186/gb-2013-14-7-r70
- Graveley BR. 2001. Alternative splicing: increasing diversity in the proteomic world. *Trends Genet* **17**: 100–107. doi:10.1016/S0168-9525(00)02176-4
- Greenberg MJ, Ostap EM. 2013. Regulation and control of myosin-I by the motor and light chain-binding domains. *Trends Cell Biol* **23**: 81–89. doi:10.1016/j.tcb.2012.10.008
- Heber S, Alekseyev M, Sze SH, Tang H, Pevzner PA. 2002. Splicing graphs and EST assembly problem. *Bioinformatics* **18**: S181–S188. doi:10.1093/bioinformatics/18.suppl_1.S181
- Herrero J, Muffato M, Beal K, Fitzgerald S, Gordon L, Pignatelli M, Vilella AJ, Searle SM, Amode R, Brent S, et al. 2016. Ensembl comparative genomics resources. *Database (Oxford)* **2016**: baw053. doi:10.1093/database/baw053
- Houdusse A, Silver M, Cohen C. 1996. A model of Ca²⁺-free calmodulin binding to unconventional myosins reveals how calmodulin acts as a regulatory switch. *Structure* **4**: 1475–1490. doi:10.1016/S0969-2126(96)00154-2
- Kelemen O, Convertini P, Zhang Z, Wen Y, Shen M, Falaleeva M, Stamm S. 2013. Function of alternative splicing. *Gene* **514**: 1–30. doi:10.1016/j.gene.2012.07.083
- Khan S, Downing KH, Molloy JE. 2019. Architectural dynamics of CaMKII–actin networks. *Biophys J* **116**: 104–119. doi:10.1016/j.bpj.2018.11.006
- Kim MS, Pinto SM, Getnet D, Nirujogi RS, Manda SS, Chaerkady R, Madugundu AK, Kelkar DS, Isserlin R, Jain S, et al. 2014. A draft map of the human proteome. *Nature* **509**: 575–581. doi:10.1038/nature13302
- Komljenovic A, Roux J, Wollbrett J, Robinson-Rechavi M, Basian FB. 2016. BgeeDB, an R package for retrieval of curated expression datasets and for gene list expression localization enrichment tests. *F1000Res* **5**: 2748. doi:10.12688/f1000research.9973.2
- Leinonen R, Sugawara H, Shumway M, International Nucleotide Sequence Database Collaboration. 2011. The sequence read archive. *Nucleic Acids Res* **39**: D19–D21. doi:10.1093/nar/gkq1019
- Lim KH, Ferraris L, Filloux ME, Raphael BJ, Fairbrother WG. 2011. Using positional distribution to identify splicing elements and predict pre-mRNA processing defects in human genes. *Proc Natl Acad Sci USA* **108**: 11093–11098. doi:10.1073/pnas.1101135108
- Liu J, Taylor DW, Krementsova EB, Trybus KM, Taylor KA. 2006. Three-dimensional structure of the myosin V inhibited state by cryo-electron tomography. *Nature* **442**: 208–211. doi:10.1038/nature04719
- Lonsdale J, Thomas J, Salvatore M, Phillips R, Lo E, Shad S, Hasz R, Walters G, Garcia F, Young N, et al. 2013. The genotype-tissue expression (GTEx) project. *Nat Genet* **45**: 580–585. doi:10.1038/ng.2653
- Louadi Z, Yuan K, Gress A, Tsoy O, Kalinina OV, Baumbach J, Kacprowski T, List M. 2020. Digger: exploring the functional role of alternative splicing in protein interactions. *Nucleic Acids Res* **49**(D1): D309–D318. doi:10.1093/nar/gkaa768
- Marti-Solano M, Crilly SE, Malinvernini D, Munk C, Harris M, Pearce A, Quon T, Mackenzie AE, Wang X, Peng J, et al. 2020. Combinatorial expression of GPCR isoforms affects signalling and drug responses. *Nature* **588**: E24. doi:10.1038/s41586-020-2999-9
- Mei W, Boatwright L, Feng G, Schnable JC, Barbazuk WB. 2017. Evolutionarily conserved alternative splicing across monocots. *Genetics* **207**: 465–480. doi:10.1534/genetics.117.300189
- Merkin J, Russell C, Chen P, Burge CB. 2012. Evolutionary dynamics of gene and isoform regulation in Mammalian tissues. *Science* **338**: 1593–1599. doi:10.1126/science.1228186
- Modrek B, Lee CJ. 2003. Alternative splicing in the human, mouse and rat genomes is associated with an increased frequency of exon creation and/or loss. *Nat Genet* **34**: 177–180. doi:10.1038/ng1159
- Nichio BT, Marchaukoski JN, Raittz RT. 2017. New tools in orthology analysis: a brief review of promising perspectives. *Front Genet* **8**: 165. doi:10.3389/fgene.2017.00165
- Park E, Pan Z, Zhang Z, Lin L, Xing Y. 2018. The expanding landscape of alternative splicing variation in human populations. *Am J Hum Genet* **102**: 11–26. doi:10.1016/ajhg.2017.11.002
- Pathan-Chhatbar S, Taft MH, Reindl T, Hundt N, Latham SL, Manstein DJ. 2018. Three mammalian tropomyosin isoforms have different regulatory effects on nonmuscle myosin-2B and filamentous β-actin *in vitro*. *J Biol Chem* **293**: 863–875. doi:10.1074/jbc.M117.806521
- Rodriguez JM, Maietta P, Ezkurdia I, Pietrelli A, Wesselink JJ, Lopez G, Valencia A, Tress ML. 2013. APPRIIS: annotation of principal and alternative splice isoforms. *Nucleic Acids Res* **41**(Database issue): D110–D117. doi:10.1093/nar/gks1058
- Salzberg SL. 2019. Next-generation genome annotation: we still struggle to get it right. *Genome Biol* **16**: 92. doi:10.1186/s13059-019-1715-2
- Scotti MM, Swanson MS. 2016. RNA mis-splicing in disease. *Nat Rev Genet* **17**: 19–32. doi:10.1038/nrg.2015.3
- Shannon P, Markiel A, Ozier O, Baliga NS, Wang JT, Ramage D, Amin N, Schwikowski B, Ideker T. 2003. Cytoscape: a software environment for integrated models of biomolecular interaction networks. *Genome Res* **13**: 2498–2504. doi:10.1101/gr.1239303
- Siepel A, Haussler D. 2005. Phylogenetic hidden Markov models. In *Statistical methods in molecular evolution* (ed. Nielsen R), pp. 325–351. Springer, New York.
- Siepel A, Bejerano G, Pedersen JS, Hinrichs AS, Hou M, Rosenbloom K, Clawson H, Spieth J, Hillier LW, Richards S, et al. 2005. Evolutionarily conserved elements in vertebrate, insect, worm, and yeast genomes. *Genome Res* **15**: 1034–1050. doi:10.1101/gr.3715005
- Sloutsky R, Stratton MM. 2020. Functional implications of CaMKII alternative splicing. *Eur J Neurosci* doi:10.1111/ejn.14761
- Sterne-Weiler T, Weatheritt RJ, Best AJ, Ha KC, Blencowe BJ. 2018. Efficient and accurate quantitative profiling of alternative splicing patterns of any complexity on a laptop. *Mol Cell* **72**: 187–200.e6. doi:10.1016/j.molcel.2018.08.018
- Sultan M, Schulz MH, Richard H, Magen A, Klingenhoff A, Scherf M, Seifert M, Borodina T, Soldatov A, Parkhomchuk D, et al. 2008. A global view of gene activity and alternative splicing by deep sequencing of the human transcriptome. *Science* **321**: 956–960. doi:10.1126/science.1160342
- Szalkowski AM. 2012. Fast and robust multiple sequence alignment with phylogeny-aware gap placement. *BMC Bioinformatics* **13**: 129. doi:10.1186/1471-2105-13-129
- Tapial J, Ha KCH, Sterne-Weiler T, Gohr A, Braunschweig U, Hermoso-Pulido A, Quesnel-Vallières M, Permyaner J, Sodaei R, Marquez Y, et al. 2017. An atlas of alternative splicing profiles and functional associations reveals new regulatory programs and genes that simultaneously express multiple major isoforms. *Genome Res* **27**: 1759–1768. doi:10.1101/gr.220962.117
- Thompson JD, Gibson TJ, Plewniak F, Jeanmougin F, Higgins DG. 1997. The CLUSTAL_X windows interface: flexible strategies for multiple sequence

Evolutionary splicing graphs for AS conservation

- alignment aided by quality analysis tools. *Nucleic Acids Res* **25**: 4876–4882. doi:10.1093/nar/25.24.4876
- Tranchevent LC, Aubé F, Dulaurier L, Benoit-Pilven C, Rey A, Poret A, Chautard E, Mortada H, Desmet FO, Chakrama FZ, et al. 2017. Identification of protein features encoded by alternative exons using exon ontology. *Genome Res* **27**: 1087–1097. doi:10.1101/gr.212696.116
- Waetzig V, Herdegen T. 2005. Context-specific inhibition of JNKs: overcoming the dilemma of protection and damage. *Trends Pharmacol Sci* **26**: 455–461. doi:10.1016/j.tips.2005.07.006
- Wang GS, Cooper TA. 2007. Splicing in disease: disruption of the splicing code and the decoding machinery. *Nat Rev Genet* **8**: 749–761. doi:10.1038/nrg2164
- Wang L, Jiang T. 1994. On the complexity of multiple sequence alignment. *J Comput Biol* **1**: 337–348. doi:10.1089/cmb.1994.1.337
- Wang P, Wu YL, Zhou TH, Sun Y, Pei G. 2000. Identification of alternative splicing variants of the β subunit of human Ca²⁺/calmodulin-dependent protein kinase II with different activities. *FEBS Lett* **475**: 107–110. doi:10.1016/S0014-5793(00)01634-3
- Wang ET, Sandberg R, Luo S, Khrebtukova I, Zhang L, Mayr C, Kingsmore SF, Schroth GP, Burge CB. 2008. Alternative isoform regulation in human tissue transcriptomes. *Nature* **456**: 470–476. doi:10.1038/nature07509
- Wang X, Codreanu SG, Wern B, Li K, Chambers MC, Liebler DC, Zhang B. 2018. Detection of proteome diversity resulted from alternative splicing is limited by trypsin cleavage specificity. *Mol Cell Proteomics* **17**: 422–430. doi:10.1074/mcp.RA117.000155
- Ward AJ, Cooper TA. 2010. The pathobiology of splicing. *J Pathol* **220**: 152–163. doi:10.1002/path.2649
- Weatheritt RJ, Sterne-Weiler T, Blencowe BJ. 2016. The ribosome-engaged landscape of alternative splicing. *Nat Struct Mol Biol* **23**: 1117–1123. doi:10.1038/nsmb.3317
- Wu S, Liu J, Reedy MC, Tregebar RT, Winkler H, Franzini-Armstrong C, Sasaki H, Lucaveche C, Goldman YE, Reedy MK, et al. 2010. Electron tomography of cryofixed, isometrically contracting insect flight muscle reveals novel actin-myosin interactions. *PLoS One* **5**: e12643. doi:10.1371/journal.pone.0012643
- Xing Y, Lee C. 2005. Assessing the application of Ka/Ks ratio test to alternatively spliced exons. *Bioinformatics* **21**: 3701–3703. doi:10.1093/bioinformatics/bti613
- Yang X, Coulombe-Huntington J, Kang S, Sheynkman GM, Hao T, Richardson A, Sun S, Yang F, Shen YA, Murray RR, et al. 2016. Widespread expansion of protein interaction capabilities by alternative splicing. *Cell* **164**: 805–817. doi:10.1016/j.cell.2016.01.029
- Yates A, Akanni W, Amode MR, Barrell D, Billis K, Carvalho-Silva D, Cummins C, Clapham P, Fitzgerald S, Gil L, et al. 2016. Ensembl 2016. *Nucleic Acids Res* **44**: D710–D716. doi:10.1093/nar/gkv1157

Received November 27, 2020; accepted in revised form June 11, 2021.

Bibliographie

- [Abi+20] Aman ABIDI et al., « Pivot-based Maximal Biclique Enumeration. », *in : IJCAI*, 2020, p. 3558-3564.
- [ABL10] Yong-Yeol AHN, James P BAGROW et Sune LEHMANN, « Link communities reveal multiscale complexity in networks », *in : nature* 466.7307 (2010), p. 761-764.
- [AC09] Reid ANDERSEN et Kumar CHELLAPILLA, « Finding dense subgraphs with size bounds », *in : International workshop on algorithms and models for the web-graph*, Springer, 2009, p. 25-37.
- [AG10] Thomas AYNAUD et Jean-Loup GUILLAUME, « Static community detection algorithms for evolving networks », *in : 8th international symposium on modeling and optimization in mobile, ad hoc, and wireless networks*, IEEE, 2010, p. 513-519.
- [Ahm+15] Nesreen K AHMED et al., « Efficient graphlet counting for large networks », *in : 2015 IEEE international conference on data mining*, IEEE, 2015, p. 1-10.
- [Alo07] Uri ALON, « Network motifs : theory and experimental approaches », *in : Nature Reviews Genetics* 8.6 (2007), p. 450-461.
- [AMS96] Noga ALON, Yossi MATIAS et Mario SZEGEDY, « The space complexity of approximating the frequency moments », *in : Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, 1996, p. 20-29.
- [AR22] Norah ALOTAIBI et Delel RHOUMA, « A review on community structures detection in time evolving social networks », *in : Journal of King Saud University-Computer and Information Sciences* 34.8 (2022), p. 5646-5662.
- [BA99] Albert-László BARABÁSI et Réka ALBERT, « Emergence of scaling in random networks », *in : science* 286.5439 (1999), p. 509-512.
- [Bar+23] Dániel L BARABÁSI et al., « Neuroscience needs Network Science », *in : Journal of Neuroscience* 43.34 (2023), p. 5989-5995.
- [Bas+21] Giulia BASSIGNANA et al., « Stepwise target controllability identifies dysregulations of macrophage networks in multiple sclerosis », *in : Network Neuroscience* 5.2 (2021), p. 337-357.

- [Bau+05] Jeffrey BAUMES et al., « Finding communities by clustering a graph into overlapping subgraphs. », *in : IADIS AC 5* (2005), p. 97-104.
- [Bau+22] Alexis BAUDIN et al., « Clique percolation method : memory efficient almost exact communities », *in : Advanced Data Mining and Applications : 17th International Conference, ADMA 2021, Sydney, NSW, Australia, February 2–4, 2022, Proceedings, Part II*, Springer, 2022, p. 113-127.
- [BBH11] Balabhaskar BALASUNDARAM, Sergiy BUTENKO et Illya V HICKS, « Clique relaxations in social network analysis : The maximum k-plex problem », *in : Operations Research 59.1* (2011), p. 133-142.
- [BDK15] Vincent D BLONDEL, Adeline DECUYPER et Gautier KRINGS, « A survey of results on mobile phone datasets analysis », *in : EPJ data science 4* (2015), p. 1-55.
- [Ben+19] Matthias BENTERT et al., « Listing all maximal k-plexes in temporal graphs », *in : Journal of Experimental Algorithmics (JEA) 24* (2019), p. 1-27.
- [BK73] Coen BRON et Joep KERBOSCH, « Algorithm 457 : finding all cliques of an undirected graph », *in : Communications of the ACM 16.9* (1973), p. 575-577.
- [Bla+20] Jovan BLANUŠA et al., « Manycore clique enumeration with fast set intersections », *in : Proceedings of the VLDB Endowment 13.12* (2020), p. 2676-2690.
- [Blo+12] Benjamin BLONDER et al., « Temporal dynamics and network analysis », *in : Methods in Ecology and Evolution 3.6* (2012), p. 958-972.
- [BMT23a] Alexis BAUDIN, Clémence MAGNIEN et Lionel TABOURIER, « Énumération efficace des cliques maximales dans les flots de liens réels massifs. », *in : 23ème conférence francophone sur l'extraction et la gestion des connaissances, EGC 2023*, 2023, p. 139-150.
- [BMT23b] Alexis BAUDIN, Clémence MAGNIEN et Lionel TABOURIER, « Faster maximal clique enumeration in large real-world link streams », *in : (fév. 2023)*.
- [Bou+18] Souâad BOUDEBZA et al., « OLCPM : An online framework for detecting overlapping communities in dynamic social networks », *in : Computer Communications 123* (2018), p. 36-51.
- [BS17] Danielle S BASSETT et Olaf SPORNS, « Network neuroscience », *in : Nature neuroscience 20.3* (2017), p. 353-364.

- [BTM23] Alexis BAUDIN, Lionel TABOURIER et Clémence MAGNIEN, « LSCPM : Communities in Massive Real-World Link Streams by Clique Percolation Method », in : *30th International Symposium on Temporal Representation and Reasoning, TIME 2023*, 2023.
- [BZ03] Vladimir BATAGELJ et Matjaz ZAVERSNIK, « An $\text{O}(\text{m})$ algorithm for cores decomposition of networks », in : *arXiv preprint cs/0310049* (2003).
- [Can+08] Julián CANDIA et al., « Uncovering individual and collective human dynamics from mobile phone records », in : *Journal of physics A : mathematical and theoretical* 41.22 (2008), p. 224015.
- [Cas+12] Arnaud CASTEIGTS et al., « Time-varying graphs and dynamic networks », in : *International Journal of Parallel, Emergent and Distributed Systems* 27.5 (2012), p. 387-408.
- [Cat+10] Ciro CATTUTO et al., « Dynamics of person-to-person interactions from distributed RFID sensor networks », in : *PloS one* 5.7 (2010), e11596.
- [CBD22] Saint-Clair CHABERT-LIDDELL, Pierre BARBILLON et Sophie DONNET, « Impact of the mesoscale structure of a bipartite ecological interaction network on its robustness through a probabilistic modeling », in : *Environmetrics* 33.2 (2022), e2709.
- [Che+10] Wei CHEN et al., « A game-theoretic framework to identify overlapping communities in social networks », in : *Data Mining and Knowledge Discovery* 21 (2010), p. 224-240.
- [Che+22] Lu CHEN et al., « Efficient maximal biclique enumeration for large sparse bipartite graphs », in : *Proceedings of the VLDB Endowment* 15.8 (2022), p. 1559-1571.
- [Che+23] Zi CHEN et al., « Index-based biclique percolation communities search on bipartite graphs », in : *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, IEEE, 2023, p. 2699-2712.
- [CN85] Norishige CHIBA et Takao NISHIZEKI, « Arboricity and subgraph listing algorithms », in : *SIAM Journal on computing* 14.1 (1985), p. 210-223.
- [Con+17] Alessio CONTE et al., « Fast enumeration of large k-plexes », in : *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, 2017, p. 115-124.
- [Con+18] Alessio CONTE et al., « D2K : scalable community detection in massive networks via small-diameter k-plexes », in : *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, p. 1272-1281.

- [Con+20] Alessio CONTE et al., « Sublinear-space and bounded-delay algorithms for maximal clique enumeration in graphs », *in : Algorithmica* 82 (2020), p. 1547-1573.
- [CP19] Raphaël CHARBEY et Christophe PRIEUR, « Stars, holes, or paths across your facebook friends : A graphlet-based characterization of many networks », *in : Network Science* 7.4 (2019), p. 476-497.
- [CRA17] Rémy CAZABET, Giulio ROSSETTI et Frédéric AMBLARD, *Dynamic community detection*, 2017.
- [CT22] Alessio CONTE et Etsushi TOMITA, « On the overall and delay complexity of the CLIQUES and Bron-Kerbosch algorithms », *in : Theoretical Computer Science* 899 (2022), p. 1-24.
- [Cui+13] Wanyun CUI et al., « Online search of overlapping communities », *in : Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, 2013, p. 277-288.
- [Dak+19] Narimene DAKICHE et al., « Tracking community evolution in social networks : A survey », *in : Information Processing & Management* 56.3 (2019), p. 1084-1102.
- [Dam14] Peter DAMASCHKE, « Enumerating maximal bicliques in bipartite graphs with favorable degree sequences », *in : Information Processing Letters* 114.6 (2014), p. 317-321.
- [DBS18] Maximilien DANISCH, Oana Denisa BALALAU et Mauro SOZIO, « Listing k-cliques in Sparse Real-World Graphs. », *in : WWW*, sous la dir. de Pierre-Antoine CHAMPIN et al., ACM, 2018, p. 589-598.
- [DCS17] Maximilien DANISCH, T-H Hubert CHAN et Mauro SOZIO, « Large scale density-friendly graph decomposition via convex programming », *in : Proceedings of the 26th International Conference on World Wide Web*, 2017, p. 233-242.
- [DDZ14] Naga Shailaja DASARI, Ranjan DESH et Mohammad ZUBAIR, « pbitMCE : A bit-based approach for maximal clique enumeration on multicore processors », *in : 2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, IEEE, 2014, p. 478-485.
- [DEV14] Bhagat Lal DUTTA, Pauline EZANNO et Elisabeta VERGU, « Characteristics of the spatio-temporal network of cattle movements in France over a 5-year period », *in : Preventive veterinary medicine* 117.1 (2014), p. 79-94.
- [DFI18] Camil DEMETRESCU, Irene FINOCCHI et Giuseppe F ITALIANO, « Dynamic graphs », *in : Handbook of Data Structures and Applications*, Chapman et Hall/CRC, 2018, p. 581-594.

- [DH73] William E DONATH et Alan J HOFFMAN, « Lower bounds for the partitioning of graphs », in : *IBM Journal of Research and Development* 17.5 (1973), p. 420-425.
- [DST18] Apurba DAS, Seyed-Vahid SANEI-MEHRI et Srikanta TIRTHAPURA, « Shared-memory parallel maximal clique enumeration », in : *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, IEEE, 2018, p. 62-71.
- [DST19] Apurba DAS, Michael SVENDSEN et Srikanta TIRTHAPURA, « Incremental maintenance of maximal cliques in a dynamic graph », in : *The VLDB Journal* 28.3 (2019), p. 351-375.
- [EL10] Tim S EVANS et Renaud LAMBIOTTE, « Line graphs of weighted networks for overlapping communities », in : *The European Physical Journal B* 77 (2010), p. 265-272.
- [ELS10] David EPPSTEIN, Maarten LÖFFLER et Darren STRASH, « Listing all maximal cliques in sparse graphs in near-optimal time », in : *International Symposium on Algorithms and Computation*, Springer, 2010, p. 403-414.
- [EP06] Nathan EAGLE et Alex PENTLAND, « Reality mining : sensing complex social systems », in : *Personal and ubiquitous computing* 10 (2006), p. 255-268.
- [ERS77] Paul ERDÖS, BL ROTHSCHILD et NM SINGHI, « Characterizing cliques in hypergraphs », in : *Ars Combinatoria* 4.H977 (1977), p. 81-118.
- [ES11] David EPPSTEIN et Darren STRASH, « Listing all maximal cliques in large sparse real-world graphs », in : *Experimental Algorithms* (2011), p. 364-375.
- [Eve+11] Brian S EVERITT et al., *Cluster analysis*, John Wiley & Sons, 2011.
- [FB14a] Julie FOURNET et Alain BARRAT, « Contact patterns among high school students », in : *PloS one* 9.9 (2014), Données disponibles à l'adresse : <http://www.sociopatterns.org/datasets/high-school-dynamic-contact-networks>, e107878.
- [FB14b] Julie FOURNET et Alain BARRAT, « Contact patterns among high school students », in : *PloS one* 9.9 (2014), Data available at : <http://www.sociopatterns.org/datasets/high-school-dynamic-contact-networks>, e107878.
- [FC12] Santo FORTUNATO et Claudio CASTELLANO, « Community Structure in Graphs », in : *Computational Complexity : Theory, Techniques, and Applications*, sous la dir. de Robert A. MEYERS, Springer New York, 2012, p. 490-512, ISBN : 978-1-4614-1800-9.

- [Fie73] Miroslav FIEDLER, « Algebraic connectivity of graphs », *in : Czechoslovak mathematical journal* 23.2 (1973), p. 298-305.
- [For10] Santo FORTUNATO, « Community detection in graphs », *in : Physics reports* 486.3-5 (2010), p. 75-174.
- [Fra+06] Eugene FRATKIN et al., « MotifCut : regulatory motifs finding with maximum density subgraphs », *in : Bioinformatics* 22.14 (2006), e150-e157.
- [Gao+11] Wei GAO et al., « Temporal and spatial evolution of brain network topology during the first two years of life », *in : PloS one* 6.9 (2011), e25278.
- [Gao+22] Zhongqiang GAO et al., « Scalable motif counting for large-scale temporal graphs », *in : 2022 IEEE 38th International Conference on Data Engineering (ICDE)*, IEEE, 2022, p. 2656-2668.
- [Gau+16] Noé GAUMONT et al., « Analysis of the temporal and structural features of threads in a mailing-list », *in : Complex Networks VII : Proceedings of the 7th Workshop on Complex Networks CompleNet 2016*, Springer, 2016, p. 107-118.
- [GBC14] Valerio GEMMETTO, Alain BARRAT et Ciro CATTUTO, « Mitigation of infectious disease at school : targeted class closure vs school closure », *in : BMC infectious diseases* 14.1 (2014), Données disponibles à l'adresse : <http://www.sociopatterns.org/datasets/primary-school-temporal-network-data>, p. 1-10.
- [GKT05] David GIBSON, Ravi KUMAR et Andrew TOMKINS, « Discovering large dense subgraphs in massive graphs », *in : Proceedings of the 31st international conference on Very large data bases*, 2005, p. 721-732.
- [GL06] Jean-Loup GUILLAUME et Matthieu LATAPY, « Bipartite graphs as models of complex networks », *in : Physica A : Statistical Mechanics and its Applications* 371.2 (2006), p. 795-813.
- [GLM13] Enrico GREGORI, Luciano LENZINI et Simone MAINARDI, « Parallel k-clique community detection on large-scale networks », *in : IEEE Transactions on Parallel and Distributed Systems* 24.8 (2013), Implémentation de l'algorithme : <https://sourceforge.net/p/cosparallel>, p. 1651-1660.
- [GN02] Michelle GIRVAN et Mark EJ NEWMAN, « Community structure in social and biological networks », *in : Proceedings of the national academy of sciences* 99.12 (2002), p. 7821-7826.
- [GNS09] Alain GÉLY, Lhouari NOURINE et Bachir SADI, « Enumeration aspects of maximal cliques and bicliques », *in : Discrete applied mathematics* 157.7 (2009), p. 1447-1459.

- [Gol84] Andrew V GOLDBERG, « Finding a maximum density subgraph », *in* : (1984).
- [Ham20] William L HAMILTON, *Graph representation learning*, Morgan & Claypool Publishers, 2020.
- [Har+16] Christopher R HARSHAW et al., « Graphprints : Towards a graph analytic method for network anomaly detection », *in* : *Proceedings of the 11th Annual Cyber and Information Security Research Conference*, 2016, p. 1-4.
- [Him+16] Anne-Sophie HIMMEL et al., « Enumerating maximal cliques in temporal graphs », *in* : *2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, IEEE, 2016, p. 337-344.
- [Him+17] Anne-Sophie HIMMEL et al., « Adapting the Bron–Kerbosch algorithm for enumerating maximal cliques in temporal graphs », *in* : *Social Network Analysis and Mining* 7.1 (2017), p. 1-16.
- [HLS09] Theus HOSSMANN, Franck LEGENDRE et Thrasyvoulos SPYROPOULOS, « From contacts to graphs : Pitfalls in using complex network analysis for dtn routing », *in* : *IEEE INFOCOM Workshops 2009*, IEEE, 2009, p. 1-6.
- [HM21] Danny HERMELIN et George MANOUSSAKIS, « Efficient enumeration of maximal induced bicliques », *in* : *Discrete Applied Mathematics* 303 (2021), p. 253-261.
- [Hol15] Petter HOLME, « Modern temporal network theory : a colloquium », *in* : *The European Physical Journal B* 88 (2015), p. 1-30.
- [Hol20] Andreas F HOLMSEN, « Large cliques in hypergraphs with forbidden substructures », *in* : *Combinatorica* 40.4 (2020), p. 527-537.
- [HS12] Petter HOLME et Jari SARAMÄKI, « Temporal networks », *in* : *Physics reports* 519.3 (2012), p. 97-125.
- [Ise+11] Lorenzo ISELLA et al., « What's in a Crowd? Analysis of Face-to-Face Behavioral Networks », *in* : *Journal of Theoretical Biology* 271.1 (2011), Data hypertext : <http://www.sociopatterns.org/datasets/hypertext-2009-dynamic-contact-network>. Data infectious : <http://www.sociopatterns.org/datasets/infectious-sociopatterns/>, p. 166-180, ISSN : 0022-5193.
- [JF16] David MP JACOBY et Robin FREEMAN, « Emerging network-based tools in movement ecology », *in* : *Trends in Ecology & Evolution* 31.4 (2016), p. 301-314.
- [Jin+22] Yan JIN et al., « On fast enumeration of maximal cliques in large graphs », *in* : *Expert Systems with Applications* 187 (2022), p. 115915.

- [JS20] Shweta JAIN et C SESHADHRI, « The power of pivoting for exact clique counting », *in : Proceedings of the 13th International Conference on Web Search and Data Mining*, 2020, p. 268-276.
- [KA12] Hyoungshick KIM et Ross ANDERSON, « Temporal node centrality in complex networks », *in : Physical Review E* 85.2 (2012), p. 026107.
- [Kar+11] Márton KARSAI et al., « Small but slow world : How network topology and burstiness slow down spreading », *in : Physical Review E* 83.2 (2011), p. 025102.
- [Kel+12] Stephen KELLEY et al., « Defining and discovering communities in social networks », *in : Handbook of optimization in complex networks : Theory and applications* (2012), p. 139-168.
- [Kit+16] Moses C KITI et al., « Quantifying social contacts in a household setting of rural Kenya using wearable proximity sensors », *in : EPJ data science* 5 (2016), p. 1-21.
- [KL70] Brian W KERNIGHAN et Shen LIN, « An efficient heuristic procedure for partitioning graphs », *in : The Bell system technical journal* 49.2 (1970), p. 291-307.
- [Kos09] Vassilis KOSTAKOS, « Temporal graphs », *in : Physica A : Statistical Mechanics and its Applications* 388.6 (2009), p. 1007-1023.
- [Kov+11] Lauri KOVANEN et al., « Temporal motifs in time-dependent networks », *in : Journal of Statistical Mechanics : Theory and Experiment* 2011.11 (2011), P11005.
- [Kov+13] Lauri KOVANEN et al., « Temporal motifs reveal homophily, gender-specific patterns, and group talk in call sequences », *in : Proceedings of the National Academy of Sciences* 110.45 (2013), p. 18070-18075.
- [Kov+19] István A KOVÁCS et al., « Network-based prediction of protein interactions », *in : Nature communications* 10.1 (2019), p. 1240.
- [KR23] Rachel KIRSCH et Jamie RADCLIFFE, « Many cliques in bounded-degree hypergraphs », *in : SIAM Journal on Discrete Mathematics* 37.3 (2023), p. 1436-1456.
- [Kri+12] Gautier KRINGS et al., « Effects of time window size and placement on the structure of an aggregated communication network », *in : EPJ Data Science* 1.1 (2012), p. 1-16.
- [Kum+08] Jussi M KUMPULA et al., « Sequential algorithm for fast clique percolation », *in : Physical Review E* 78.2 (2008), Implémentation de l'algorithme : <https://github.com/CxAalto/scp>, p. 026109.
- [Kun13] Jérôme KUNEGIS, « KONECT – The Koblenz Network Collection », *in : Proc. Int. Conf. on World Wide Web Companion*, Données disponibles à l'adresse : <http://konect.cc/networks>, 2013, p. 1343-1350.

- [Lat08] Matthieu LATAPY, « Main-memory triangle computations for very large (sparse (power-law)) graphs », *in : Theoretical computer science* 407.1-3 (2008), p. 458-473.
- [LCF19] Yannick Léo, Christophe CRESPELLE et Eric FLEURY, « Non-altering time scales for aggregation of dynamic networks into series of graphs », *in : Computer Networks* 148 (2019), p. 108-119.
- [Léc+23] Fabrice LÉCUYER et al., « Tailored vertex ordering for faster triangle listing in large graphs », *in : 2023 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, SIAM, 2023, p. 77-85.
- [Lee+10] Victor E LEE et al., « A survey of algorithms for dense subgraph discovery », *in : Managing and mining graph data* (2010), p. 303-336.
- [Les+17] Brenton LESSLEY et al., « Maximal clique enumeration with data-parallel primitives », *in : 2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV)*, IEEE, 2017, p. 16-25.
- [LFK09] Andrea LANCICHINETTI, Santo FORTUNATO et János KERTÉSZ, « Detecting the overlapping and hierarchical community structure in complex networks », *in : New journal of physics* 11.3 (2009), p. 033015.
- [LH08] Jure LESKOVEC et Eric HORVITZ, « Planetary-scale views on a large instant-messaging network », *in : Proceedings of the 17th international conference on World Wide Web*, 2008, p. 915-924.
- [Li+07] Jinyan LI et al., « Maximal biclique subgraphs and closed pattern pairs of the adjacency matrix : A one-to-one correspondence and mining algorithms », *in : IEEE Transactions on Knowledge and Data Engineering* 19.12 (2007), p. 1625-1637.
- [Li+17] Aming LI et al., « The fundamental advantages of temporal networks », *in : Science* 358.6366 (2017), p. 1042-1046.
- [Li+20] Rong-Hua LI et al., « Ordering heuristics for k -clique listing », *in : PVLDB*, 2020.
- [Liu+22] Penghang LIU et al., « Temporal motifs in patent opposition and collaboration networks », *in : Scientific reports* 12.1 (2022), p. 1917.
- [LK03] David LIBEN-NOWELL et Jon KLEINBERG, « The link prediction problem for social networks », *in : Proceedings of the twelfth international conference on Information and knowledge management*, 2003, p. 556-559.
- [LK14] Jure LESKOVEC et Andrej KREVL, *SNAP Datasets : Stanford Large Network Dataset Collection*, <http://snap.stanford.edu/data>, juin 2014.

- [LKF05] Jure LESKOVEC, Jon KLEINBERG et Christos FALOUTSOS, « Graphs over time : densification laws, shrinking diameters and possible explanations », in : *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, Données disponibles à l'adresse : <https://snap.stanford.edu/data/as-733.html>, 2005, p. 177-187.
- [LKF07] Jure LESKOVEC, Jon KLEINBERG et Christos FALOUTSOS, « Graph evolution : Densification and shrinking diameters », in : *ACM transactions on Knowledge Discovery from Data (TKDD)* 1.1 (2007), 2-es.
- [LLR80] Eugene L. LAWLER, Jan Karel LENSTRA et AHG RINNOOY KAN, « Generating all maximal independent sets : NP-hardness and polynomial-time algorithms », in : *SIAM Journal on Computing* 9.3 (1980), p. 558-565.
- [Lur+20] Miguel LURGI et al., « Geographical variation of multiplex ecological networks in marine intertidal communities », in : *Ecology* 101.11 (2020), e03165.
- [LVM18] Matthieu LATAPY, Tiphaine VIARD et Clémence MAGNIEN, « Stream graphs and link streams for the modeling of interactions over time », in : *Social Network Analysis and Mining* 8.1 (2018), p. 1-29.
- [LW20] Erica LL LIU et Jian WANG, « The maximum number of cliques in hypergraphs without large matchings », in : *arXiv preprint arXiv :2005.01080* (2020).
- [LZ11] Linyuan LÜ et Tao ZHOU, « Link prediction in complex networks : A survey », in : *Physica A : statistical mechanics and its applications* 390.6 (2011), p. 1150-1170.
- [MA03] Shmoolik MANGAN et Uri ALON, « Structure and function of the feed-forward loop network motif », in : *Proceedings of the National Academy of Sciences* 100.21 (2003), p. 11980-11985.
- [MCF14] Lucie MARTINET, Christophe CRESPELLE et Eric FLEURY, « Dynamic contact network analysis in hospital wards », in : *Complex Networks V : Proceedings of the 5th Workshop on Complex Networks CompleNet 2014*, Springer, 2014, p. 241-249.
- [MFB15] Rossana MASTRANDREA, Julie FOURNET et Alain BARRAT, « Contact patterns in a high school : a comparison between data collected using wearable sensors, contact diaries and friendship surveys », in : *PloS one* 10.9 (2015), Données disponibles à l'adresse : <https://journals.plos.org/plosone/article?id=10.1371%2Fjournal.pone.0136497#sec012>, e0136497.

- [MGH13] Aaron F. MCDAID, Derek GREENE et Neil HURLEY, *Normalized Mutual Information to evaluate overlapping community finding algorithms*, <https://arxiv.org/abs/1110.2515>, 2013, arXiv : 1110.2515 [physics.soc-ph].
- [Mil+02] Ron MILO et al., « Network motifs : simple building blocks of complex networks », in : *Science* 298.5594 (2002), p. 824-827.
- [Mis+08] Alan MISLOVE et al., « Growth of the flickr social network », in : *Proceedings of the first workshop on Online social networks*, Data available at : <http://konect.cc/networks/flickr-growth>, 2008, p. 25-30.
- [Mis09] Alan MISLOVE, « Online Social Networks : Measurement, Analysis, and Applications to Distributed Information Systems », Données disponibles à l'adresse : <https://socialnetworks.mpi-sws.org/data-wosn2008.html>, thèse de doct., Rice University, Department of Computer Science, mai 2009.
- [MM65] John W MOON et Leo MOSER, « On cliques in graphs », in : *Israel journal of Mathematics* 3.1 (1965), p. 23-28.
- [MSK12] Ali MASOUDI-NEJAD, Falk SCHREIBER et Zahra Razaghi Moghadam KASHANI, « Building blocks of biological networks : a review on major network motif discovery algorithms », in : *IET systems biology* 6.5 (2012), p. 164-174.
- [Nau16] Kevin A NAUDÉ, « Refined pivot selection for maximal clique enumeration in graphs », in : *Theoretical Computer Science* 613 (2016), p. 28-37.
- [Nep+08] Tamás NEPUSZ et al., « Fuzzy communities and the concept of bridgeness in complex networks », in : *Physical Review E* 77.1 (2008), p. 016107.
- [New03] Mark EJ NEWMAN, « The structure and function of complex networks », in : *SIAM review* 45.2 (2003), p. 167-256.
- [NG04] Mark EJ NEWMAN et Michelle GIRVAN, « Finding and evaluating community structure in networks », in : *Physical review E* 69.2 (2004), p. 026113.
- [Ngu+11] Nam P NGUYEN et al., « Overlapping communities in dynamic networks : their detection and mobile applications », in : *Proceedings of the 17th annual international conference on Mobile computing and networking*, 2011, p. 85-96.
- [Pal+05] Gergely PALLA et al., « Uncovering the overlapping community structure of complex networks in nature and society », in : *Nature* 435.7043 (2005), Implémentation de l'algorithme : <http://www.cfinder.org/>, p. 814-818.

- [Pan+14] Gang PAN et al., « Online community detection for large complex networks », *in : PloS one* 9.7 (2014), e102799.
- [PBL17] Ashwin PARANJAPE, Austin R BENSON et Jure LESKOVEC, « Motifs in temporal networks », *in : Proceedings of the tenth ACM international conference on web search and data mining*, 2017, p. 601-610.
- [PBV07] Gergely PALLA, Albert-László BARABÁSI et Tamás VICSEK, « Quantifying social group evolution », *in : Nature* 446.7136 (2007), p. 664-667.
- [PF22] Charley PRESIGNY et Fabrizio De Vico FALLANI, « Colloquium : Multiscale modeling of brain network organization », *in : Reviews of Modern Physics* 94.3 (2022), p. 031002.
- [PL06] Anuphap PRACHUMWAT et Wen-Hsiung LI, « Protein function, connectivity, and duplicability in yeast », *in : Molecular biology and evolution* 23.1 (2006), p. 30-39.
- [POC09a] Pietro PANZARASA, Tore OPSAHL et Kathleen M CARLEY, « Patterns and dynamics of users' behavior and interaction : Network analysis of an online community », *in : Journal of the American Society for Information Science and Technology* 60.5 (2009), p. 911-932.
- [POC09b] Pietro PANZARASA, Tore OPSAHL et Kathleen M CARLEY, « Patterns and dynamics of users' behavior and interaction : Network analysis of an online community », *in : Journal of the American Society for Information Science and Technology* 60.5 (2009), Données disponibles à l'adresse : <http://snap.stanford.edu/data/CollegeMsg.html>, p. 911-932.
- [Pot97] Alex POTHEN, « Graph partitioning algorithms with applications to scientific computing », *in : Parallel Numerical Algorithms*, Springer, 1997, p. 323-368.
- [Prž07] Nataša PRŽULJ, « Biological network comparison using graphlet degree distribution », *in : Bioinformatics* 23.2 (2007), e177-e183.
- [PSS10] Teresa M PRZYTYCKA, Mona SINGH et Donna K SLONIM, « Toward the dynamic interactome : it's about time », *in : Briefings in bioinformatics* 11.1 (2010), p. 15-29.
- [PSV17] Ali PINAR, Comandur SESADHRI et Vaidyanathan VISHAL, « Escape : Efficiently counting all 5-vertex subgraphs », *in : Proceedings of the 26th international conference on world wide web*, 2017, p. 1431-1440.
- [PTL19] Aurore PAYEN, Lionel TABOURIER et Matthieu LATAPY, « Spreading dynamics in a cattle trade network : size, speed, typical profile and consequences on epidemic control strategies », *in : PLoS One* 14.6 (2019), e0217972.

- [PYB13] Jeffrey PATTILLO, Nataly YOUSSEF et Sergiy BUTENKO, « On clique relaxation models in network analysis », *in : European Journal of Operational Research* 226.1 (2013), p. 9-18.
- [RA15] Ryan A. ROSSI et Nesreen K. AHMED, « The Network Data Repository with Interactive Graph Analytics and Visualization », *in : AAAI*, Données disponibles à l'adresse : <https://networkrepository.com/temporal-networks.php>, 2015.
- [RAK07] Usha Nandini RAGHAVAN, Réka ALBERT et Soundar KUMARA, « Near linear time algorithm to detect community structures in large-scale networks », *in : Physical review E* 76.3 (2007), p. 036106.
- [Rav+02] Erzsébet RAVASZ et al., « Hierarchical organization of modularity in metabolic networks », *in : science* 297.5586 (2002), p. 1551-1555.
- [RDN23] Aurora ROSSI, Samuel DESLAURIERS-GAUTHIER et Emanuele NATALE, *Temporal Brain Networks Database*, The dataset contains a collection of temporal brain networks. The networks are obtained from resting-state fMRI data of 1047 subjects from the Human Connectome Project (HCP)., mar. 2023, DOI : 10.57745/PR8VUV, URL : <https://hal.science/hal-04011895>.
- [Rib+21] Pedro RIBEIRO et al., « A survey on subgraph counting : concepts, algorithms, and applications to network motifs and graphlets », *in : ACM Computing Surveys (CSUR)* 54.2 (2021), p. 1-36.
- [RMH12] Fergal REID, Aaron McDAID et Neil HURLEY, « Percolation computation in complex networks », *in : Advances in Social Networks Analysis and Mining (ASONAM), 2012 IEEE/ACM International Conference on*, Implémentation de l'algorithme : <https://sites.google.com/site/cliqueperccomp/home>, IEEE, 2012, p. 274-281.
- [Ros+14] Ryan A ROSSI et al., « Fast maximum clique algorithms for large graphs », *in : Proceedings of the 23rd International Conference on World Wide Web*, 2014, p. 365-366.
- [Ros+17] Giulio ROSSETTI et al., « Tiles : an online algorithm for community discovery in dynamic social networks », *in : Machine Learning* 106 (2017), p. 1213-1241.
- [RPB13] Bruno RIBEIRO, Nicola PERRA et Andrea BARONCELLI, « Quantifying the effect of temporal resolution on time-varying networks », *in : Scientific reports* 3.1 (2013), p. 3006.
- [San+11] Nicola SANTORO et al., « Time-varying graphs and social network analysis : Temporal indicators and metrics », *in : arXiv preprint arXiv:1102.0629* (2011).

- [SC11] John SCOTT et Peter J CARRINGTON, *The SAGE handbook of social network analysis*, SAGE publications, 2011.
- [Sei83] Stephen B SEIDMAN, « Network structure and minimum degree », *in : Social networks* 5.3 (1983), p. 269-287.
- [SF78] Stephen B SEIDMAN et Brian L FOSTER, « A graph-theoretic generalization of the clique concept », *in : Journal of Mathematical sociology* 6.1 (1978), p. 139-154.
- [Ste+11] Juliette STEHLÉ et al., « Simulation of an SEIR infectious disease model on the dynamic contact network of conference attendees », *in : BMC medicine* 9.1 (2011), p. 1-15.
- [Sun+17] Shengli SUN et al., « Mining maximal cliques on dynamic graphs efficiently by local strategies », *in : 2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, IEEE, 2017, p. 115-118.
- [Sun+20] Bintao SUN et al., « Kclist++ : A simple algorithm for finding k-clique densest subgraphs in large graphs », *in : Proceedings of the VLDB Endowment (PVLDB)* (2020).
- [SWG16] Ingo SCHOLTES, Nicolas WIDER et Antonios GARAS, « Higher-order aggregate networks in the analysis of temporal networks : path structures and centralities », *in : The European Physical Journal B* 89 (2016), p. 1-15.
- [SZ19] Sandor SZABO et Bogdan ZAVALNIJ, « Reducing hypergraph coloring to clique search », *in : Discrete Applied Mathematics* 264 (2019), p. 196-207.
- [ZSC17] Aravind SANKAR, Xinyang ZHANG et Kevin Chen-Chuan CHANG, « Motif-based convolutional neural network on graphs », *in : arXiv preprint arXiv :1711.05697* (2017).
- [Tan+11] John TANG et al., « Exploiting temporal complex network metrics in mobile malware containment », *in : 2011 IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*, IEEE, 2011, p. 1-9.
- [Tar79] Robert Endre TARJAN, « A class of algorithms which require nonlinear time to maintain disjoint sets », *in : Journal of computer and system sciences* 18.2 (1979), p. 110-127.
- [Tso+13] Charalampos TSOURAKAKIS et al., « Denser than the densest subgraph : extracting optimal quasi-cliques with quality guarantees », *in : Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2013, p. 104-112.

- [Tso15] Charalampos TSOURAKAKIS, « The k-clique densest subgraph problem », in : *Proceedings of the 24th international conference on world wide web*, 2015, p. 1122-1132.
- [TTT06] Etsushi TOMITA, Akira TANAKA et Haruhisa TAKAHASHI, « The worst-case time complexity for generating all maximal cliques and computational experiments », in : *Theoretical computer science* 363.1 (2006), p. 28-42.
- [UKA+04] Takeaki UNO, Masashi KIYOMI, Hiroki ARIMURA et al., « LCM ver. 2 : Efficient mining algorithms for frequent/closed/maximal itemsets », in : *Fimi*, t. 126, 2004.
- [Ula04] Robert E ULANOWICZ, « Quantitative methods for ecological network analysis », in : *Computational biology and chemistry* 28.5-6 (2004), p. 321-339.
- [Van+13] Philippe VANHEMS et al., « Estimating potential infection transmission routes in hospital wards using wearable proximity sensors », in : *PloS one* 8.9 (2013), Données disponibles à l'adresse : <http://www.sociopatterns.org/datasets/hospital-ward-dynamic-contact-network/>, e73970.
- [VL14] Jordan VIARD et Matthieu LATAPY, « Identifying roles in an IP network with temporal and structural density », in : *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, IEEE, 2014, p. 801-806.
- [VLM16] Tiphaine VIARD, Matthieu LATAPY et Clémence MAGNIEN, « Computing maximal cliques in link streams », in : *Theoretical Computer Science* 609 (2016), p. 245-252.
- [VML18] Tiphaine VIARD, Clémence MAGNIEN et Matthieu LATAPY, « Enumerating maximal cliques in link streams with durations », in : *Information Processing Letters* 133 (2018), p. 44-48.
- [WCT21] Yi-Wen WEI, Wei-Mei CHEN et Hsin-Hung TSAI, « Accelerating the Bron-Kerbosch algorithm for maximal clique enumeration using GPUs », in : *IEEE Transactions on Parallel and Distributed Systems* 32.9 (2021), p. 2352-2366.
- [WF94] Stanley WASSERMAN et Katherine FAUST, « Social network analysis : Methods and applications », in : (1994).
- [WFZ16] Klaus WEHMUTH, Éric FLEURY et Artur ZIVIANI, « MultiAspect Graphs : Algebraic representation and algorithms », in : *Algorithms* 10.1 (2016), p. 1.

- [WJ94] Lusheng WANG et Tao JIANG, « On the complexity of multiple sequence alignment », *in : Journal of computational biology* 1.4 (1994), p. 337-348.
- [WM16] Matthew J WILLIAMS et Mirco MUSOLESI, « Spatio-temporal networks : reachability, centrality and robustness », *in : Royal Society open science* 3.6 (2016), Données disponibles à l'adresse : <https://datadryad.org/stash/dataset/doi:10.5061/dryad.3p27r>, p. 160196.
- [Won+12] Elisabeth WONG et al., « Biological network motif detection : principles and practice », *in : Briefings in bioinformatics* 13.2 (2012), p. 202-215.
- [WP07] Bin WU et Xin PEI, « A parallel algorithm for enumerating all the maximal k-plexes », *in : Pacific-Asia conference on knowledge discovery and data mining*, Springer, 2007, p. 476-483.
- [XKS13] Jierui XIE, Stephen KELLEY et Boleslaw K SZYMANSKI, « Overlapping community detection in networks : The state-of-the-art and comparative study », *in : Acm computing surveys (csur)* 45.4 (2013), p. 43.
- [XS11] Jierui XIE et Boleslaw K SZYMANSKI, « Community detection using a neighborhood strength driven label propagation algorithm », *in : 2011 IEEE Network Science Workshop*, IEEE, 2011, p. 188-195.
- [Yan+14] Dingqi YANG et al., « Modeling user activity preference by leveraging user spatial temporal characteristics in LBSNs », *in : IEEE Transactions on Systems, Man, and Cybernetics : Systems* 45.1 (2014), Data available at : <https://sites.google.com/site/yangdingqi/home/foursquare-dataset>, p. 129-142.
- [YJ23] Ho Yin YUEN et Jesper JANSSON, « Normalized L3-based link prediction in protein–protein interaction networks », *in : BMC bioinformatics* 24.1 (2023), p. 59.
- [YL19] Ting YU et Mengchi LIU, « A memory efficient maximal clique enumeration method for sparse graphs with a parallel implementation », *in : Parallel Computing* 87 (2019), p. 46-59.
- [Yu+20] En-Yu YU et al., « Identifying critical nodes in temporal networks by network embedding », *in : Scientific reports* 10.1 (2020), p. 12494.
- [Zea+21] Diego Javier ZEA et al., « Assessing conservation of alternative splicing with evolutionary splicing graphs », *in : Genome Research* 31.8 (2021), p. 1462-1473.
- [Zel+18] Rowan ZELLERS et al., « Neural motifs : Scene graph parsing with global context », *in : Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, p. 5831-5840.

- [Zha+12] Xiaohan ZHAO et al., « Multi-scale dynamics in a massive online social network », in : *Proceedings of the 2012 Internet Measurement Conference*, 2012, p. 171-184.
- [Zha+14] Yun ZHANG et al., « On finding bicliques in bipartite graphs : a novel algorithm and its application to the integration of diverse biological data types », in : *BMC bioinformatics* 15 (2014), p. 1-18.
- [Zho+20] Yi ZHOU et al., « Enumerating maximal k-plexes with worst-case time guarantee », in : *Proceedings of the AAAI Conference on Artificial Intelligence*, t. 34, 03, 2020, p. 2442-2449.
- [ZWZ07] Shihua ZHANG, Rui-Sheng WANG et Xiang-Sun ZHANG, « Identification of overlapping community structure in complex networks using fuzzy c-means clustering », in : *Physica A : Statistical Mechanics and its Applications* 374.1 (2007), p. 483-490.