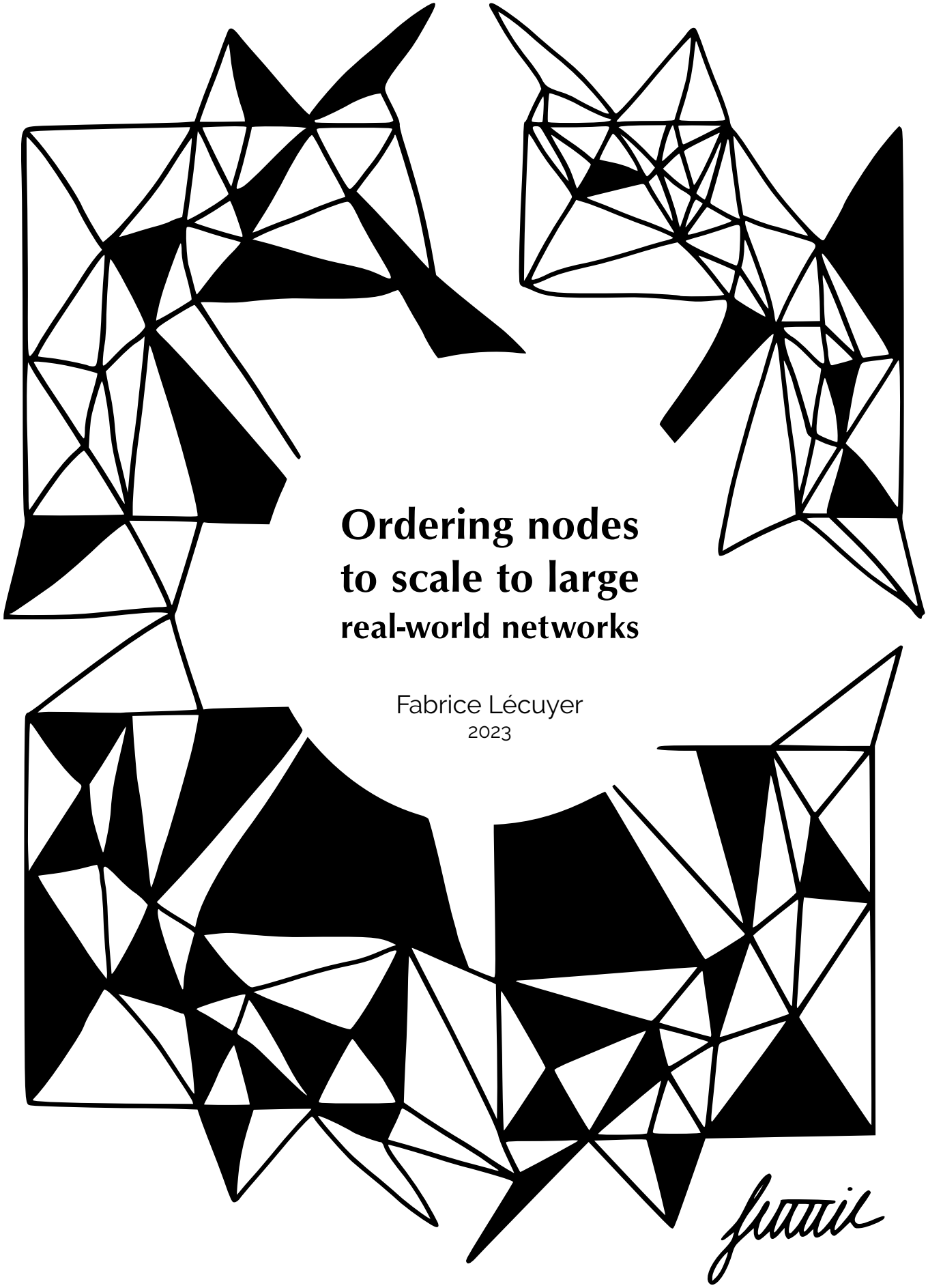


Ordering nodes to scale to large real-world networks

Fabrice Lécuyer

2023

Thèse de doctorat de Sorbonne Université
Laboratoire d'informatique LIP6
École doctorale informatique, télécommunications et électronique de Paris (EDITE)

Présentée par
Fabrice LÉCUYER

sur ses travaux de thèse intitulés

Ordonner les nœuds pour passer à l'échelle sur les grands réseaux réels

le 6 juillet 2023 à 14h, amphithéâtre 24, campus de Jussieu, Paris

devant un jury composé de

Lionel TABOURIER	directeur	Sorbonne Université, CNRS, LIP6
Clémence MAGNIEN	directrice	Sorbonne Université, CNRS, LIP6
Marco BRESSAN	rapporteur	Université de Milan
Laurent VIENNOT	rapporteur	IRIF, Université Paris Cité, INRIA
Claire HANEN	examinatrice	Sorbonne Université, CNRS, LIP6
David COUDERT	examinateur	INRIA Sophia
Rémy CAZABET	examinateur	LIRIS, Université Lyon 1

pour Katherine

Remerciements

Le bon déroulement de ces années de thèse a nécessité le concours de nombreuses personnes, tant sur l'aspect professionnel et scientifique que sur l'aspect humain et personnel.

Merci tout d'abord à Lionel et Clémence, qui ont endossé ce rôle de supervision imprévu en y investissant tout le temps nécessaire. Notre bonne entente a été à la base de tout ce travail, et ce manuscrit n'aurait pas tenu debout sans vos précieuses relectures. Je remercie aussi les rapporteurs, Marco et Laurent, pour leur lecture et leurs commentaires dans des délais serrés. Merci aux autres membres du jury qui ont accepté d'assister à la soutenance, et à toutes les personnes présentes ce jour, sur place ou par vos pensées.

Le travail n'aurait pas été aussi fluide sans l'ambiance animée et complice qui règne dans l'équipe ComplexNetworks. Merci Alexis, c'était précieux de pouvoir avancer pas-à-pas avec toi dans toutes les étapes et épreuves de la thèse, courage tu es presque au bout aussi. Merci Esteban pour les pauses café, les projets d'électronique, et les analyses sur les mondes viciés de la politique ou de la recherche. Merci Mehdi pour l'addiction aux rotations, les discussions d'histoire des religions, et tous les gâteaux. Et bien sûr merci à vous le reste de l'équipe, pour le papayoo, la géopolitique de la cantine, les manifs et une saine relation au travail.

D'autres équipes m'ont aussi ouvert leurs portes pour des collaborations, ce qui m'a permis de voir d'autres manières de s'organiser et m'a montré un échantillon de la diversité des relations humaines qui se cachent derrière la production scientifique. Merci à l'équipe du CRI, notamment Marc et Chakresh, pour m'avoir inclus dans vos travaux interdisciplinaires mais aussi dans vos célébrations et vos voyages mémorables. Amritsar will forever be a fond memory of shared fun and mind-blowing discovery. Thanks to the Systopia lab and especially to Margo, for your warm welcome at your offices, I could not imagine a better way of meeting fellow researchers abroad; I hope that we will meet again and work together in the future. Merci également aux profs de toute ma scolarité pour avoir patiemment éveillé nos curiosités et dompté nos rebellions.

Les amitiés d'école ont été essentielles pour partager les anecdotes de ce début de carrière académique et les incertitudes face au futur. Merci à la team rattrapages, vous êtes les plus brillantes des personnes qui se prennent pas au sérieux. Merci Quentin pour cette fête de l'huma tellement marquante que je vais vouloir y retourner tous les ans. Merci à vous trois du bobilehome étendu, on partage si profondément nos convictions qu'on peut discuter et rire de tout librement. Clarge, tu es ma référence en littérature et en lapins, il faut absolument qu'on s'appelle pour discuter mille nouveaux projets de startup. Sestras, j'espère qu'on pourra prévoir un tour du monde à vélo quand on aura fini nos thèses, ou au moins un week-end à Lyon, avant de se faire institutionnaliser.

Revenir à Paris en 2020 aurait pu être déprimant sans les cercles qui y étaient déjà. Merci les bails pour cette fidélité qu'on arrive à maintenir depuis bien dix ans, il n'y a rien de plus réconfortant qu'une petite soirée avec vous comme d'hab. Merci Florence pour l'illustration et Margaux pour les conseils en typo, la couverture fait tellement d'effet qu'il n'y a pas besoin de lire la suite. Pour rythmer les semaines et les semestres, il y a eu cette académie incroyable : merci Alfredo pour les ragots et les dîners, merci Célia pour ton énergie et la barre beaucoup trop haute en terme de soutenance de thèse, et merci à vous les choristes (pas aaah) qui mettez l'ambiance. Merci aussi à toutes les personnes qui viennent dire bonjour pendant vos passages à Paris.

Au plus proche, je voulais vous remercier, mes parents, pour vos valeurs de partage total et de débrouille, pour montrer qu'à tout âge on ne sait pas quoi faire plus tard, pour vos petites visites à Paris. Merci à mes frères, Mathias mon inspiration dans la recherche, Arnaud mon inspiration hors de la recherche, et après on s'étonne que j'aie du mal à décider ; les journées qu'on passe ensemble et notre entente qui perdure me sont précieuses. Merci à Joséphine, Eléa et Zélie pour votre accueil joyeux et généreux, et les liens que l'on tisse malgré la distance.

Thank you, Katherine, for following every day of these three years, making decisions for me, teaching me your language better than any teacher could, and allowing for such a smooth and homely daily life. Whatever comes next, I have loved this time with you and I admire what you have achieved.

Résumé en français

Ordonner les nœuds pour passer à l'échelle sur les grands réseaux réels

Fabrice Lécuyer

2023

Introduction

L'objectif de cette thèse est d'utiliser les outils de l'informatique théorique pour améliorer les algorithmes en pratique. Un algorithme est un ensemble d'instructions à suivre pour répondre à une question à partir de certaines données d'entrée. Les algorithmes sont généralement conçus pour les ordinateurs, mais les humains les utilisent également. Prenons l'exemple de la méthode utilisée à l'école primaire pour additionner deux grands nombres : notre prof a dit d'additionner les chiffres les plus à droite, d'écrire les unités du résultat et de reporter la retenue si nécessaire ; puis de répéter l'opération avec le chiffre suivant.

Le travail présenté dans ce manuscrit se concentre sur les algorithmes qui traitent les données sous forme de *graphes* plutôt que d'entiers comme dans l'algorithme d'addition ci-dessus. Un graphe est un objet mathématique qui représente des éléments en interaction ; un élément est appelé *nœud* ou *sommet*, et deux éléments qui interagissent sont liés par une *arête*. La Figure 1.1 montre un exemple de graphe. L'informatique s'intéresse depuis longtemps aux propriétés de ces objets et a conçu des algorithmes théoriques pour des graphes arbitraires : vérifier si le graphe entier est connecté, trouver les chemins les plus courts ou le plus grand ensemble de nœuds interconnectés, etc. Des modèles de graphes aléatoires où les nœuds ont une certaine probabilité d'interagir ont aussi été étudiés. Cependant, les graphes constituent également une structure simple pour décrire un certain nombre de situations réelles : le commerce international, les embouteillages, la collaboration scientifique... Les personnes expertes des domaines correspondants peuvent collecter ces données et les partager avec d'autres sous la forme d'un graphe. L'étude de ces *réseaux réels* diffère de l'étude des modèles mathématiques de graphes : ils ont des

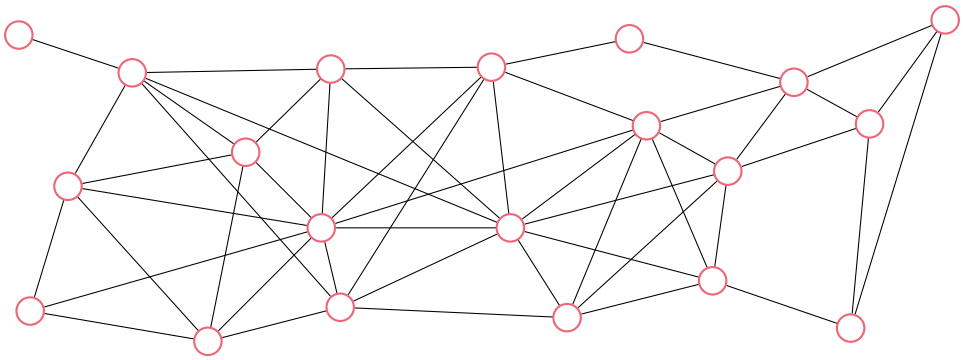


Figure 1: **Example of graph** with 20 nodes (red) and 50 edges (black).

propriétés spécifiques dues à la situation dont ils sont issus, mais ces propriétés ne sont pas connues à l'avance.

Le principal défi associé aux réseaux du monde réel est leur taille. Notre capacité collective à collecter et à organiser des données permet de disposer de graphes de plus en plus grands : les réseaux sociaux en ligne peuvent tracer des millions de membres et leurs milliards d'interactions, des trillions de courriels sont envoyés chaque année entre des milliards d'internautes, et les biologistes travaillent activement à cartographier les 10^{14} (cent mille milliards) de connexions synaptiques d'un cerveau humain.

Combien de temps faut-il à un algorithme pour traiter de tels graphes ? En d'autres termes, est-il possible de concevoir des algorithmes qui passent à l'échelle des plus grands réseaux réels ? La relation entre la taille du graphe et le temps d'exécution de l'algorithme est appelée *complexité en temps*. Pour les grands graphes, l'objectif est d'avoir une complexité linéaire, soit un temps à peu près proportionnel à la taille du graphe, où la taille est le nombre de nœuds ou d'arêtes du graphe. Pour certaines questions classiques, la communauté scientifique ne connaît malheureusement que des algorithmes qui répondent en temps exponentiel dans le pire des cas. Le pire des cas est souvent étudié en algorithmique pour qualifier la difficulté d'un problème. Cependant, comme mentionné précédemment, les réseaux réels ont des propriétés spécifiques qui les distinguent du pire cas, à tel point qu'il a été observé que de nombreux algorithmes ayant une complexité élevée dans le pire cas se comportent de manière quasi linéaire sur les réseaux réels. D'où les contraintes générales de cette thèse pour le passage à l'échelle : en termes de complexité, les algorithmes doivent avoir une complexité théorique raisonnable (non exponentielle) sur les classes de graphes considérées ; en termes de temps d'exécution, ils doivent évoluer de manière quasi-linéaire et prendre moins de quelques heures pour traiter un graphe avec des milliards d'arêtes sur un ordinateur portable standard (à condition qu'il y ait suffisamment de mémoire).

Pour résoudre le problème de passage à l'échelle, cette thèse propose d'*ordonner les nœuds*. Ordonner les nœuds d'un graphe dans un ordre pertinent s'est avéré essentiel pour résoudre divers problèmes dans les grands graphes. Pour nous convaincre de l'importance de l'ordre, reprenons l'exemple des additions : si nous devons calculer mentalement 19+7+3, dans quel ordre allons-nous le faire ? Commencer par 19+7 n'est pas évident, alors que nous pouvons reconnaître directement 7+3=10, après quoi ajouter 19 est plus simple. Dans les deux cas, le résultat est 29 et nous parvenons à le trouver. Cependant, l'ordre dans lequel nous traitons les éléments a une influence sur le temps total de l'algorithme. Le même phénomène se produit dans les algorithmes de graphes sur ordinateur : le choix d'un ordre approprié des nœuds peut rendre l'exécution plus rapide ou

le résultat plus précis. D'où la question qui nous guidera tout au long de ce manuscrit :

Comment l'ordre des nœuds
peut-il nous aider à passer à l'échelle
sur les grands réseaux réels ?

Pour répondre à cette question, la première étape consiste à identifier les problèmes d'algorithmique de graphes qui peuvent bénéficier d'un réordonnancement des nœuds. Les graphes représentent normalement des éléments en interaction de manière non ordonnée : il n'y a pas de concept naturel de premier nœud dans un graphe. C'est pourquoi la plupart des algorithmes de graphes ne semblent pas, à première vue, dépendre de l'ordre des nœuds. Toutefois, les algorithmes traitent les données une par une, ce qui fait émerger un ordre implicite : il y a un premier nœud pour l'algorithme. Le défi est de comprendre pour quels algorithmes cet ordre est important et quels gains peut apporter un réordonnancement : temps, qualité, garantie mathématique...

Une fois qu'un problème pertinent a été identifié, la tâche principale consiste à concevoir un ordre qui améliore les algorithmes correspondants. Il existe heureusement un large répertoire d'ordres auxquels on peut faire appel, dont beaucoup ont été conçus en tenant compte des contraintes de complexité. Parmi eux, le plus polyvalent est sans doute l'ordre des degrés : les nœuds sont classés en fonction du nombre d'arêtes dans lesquelles ils sont impliqués. De nouveaux problèmes peuvent réutiliser un tel ordre ou l'adapter à leurs propres caractéristiques. Il est également courant de définir de nouveaux ordres, auquel cas une attention particulière est portée au temps de calcul lorsqu'il s'agit d'applications pratiques.

Pour valider la pertinence d'un ordre, il est essentiel de disposer à la fois de preuves mathématiques et de résultats expérimentaux. Les preuves mathématiques permettent d'ancrer la méthode d'ordonnancement dans la théorie existante et de garantir le comportement des algorithmes en toutes circonstances : par exemple, prouver qu'un ordre spécifique réduit la complexité temporelle d'un algorithme donne un nouvel aperçu du problème, ce qui peut conduire à d'autres résultats théoriques. Les résultats expérimentaux mettent l'accent sur les cas d'utilisation où la méthode d'ordonnancement a un impact quantitatif et indiquent des applications possibles dans le monde réel.

Résumé des contributions

Compte tenu de ces trois tâches que sont l'identification du problème, la conception de l'ordre et la validation expérimentale, voici l'organisation du manuscrit et nos principales contributions :

Chapitre 2
Préliminaires
De la théorie des graphes à l'analyse des réseaux

Ce chapitre introduit les nations d'informatique et les notations utilisées tout au long de la thèse. Il présente d'abord la théorie des graphes avec des notions de complexité algorithmique et de structures de données. Il décrit ensuite les défis associés à l'étude des graphes réels et présente les sources et les propriétés des jeux de données utilisés dans les expériences. Il aborde également la question des fluctuations des mesures de temps dues aux contraintes matérielles.

Chapitre 3
Revue des ordres existants

Ce chapitre est une revue des méthodes d'ordonnancement des nœuds existantes dans un éventail de domaines d'application. La contribution est une liste étendue des ordres qui apparaissent sous différents noms et formes dans la littérature, ainsi qu'une classification des principaux mécanismes qui requièrent des ordres et des méthodes algorithmiques utilisées pour les calculer. Nous abordons également les cas où les ordres représentent d'autres problèmes de la théorie des graphes. Nous examinons en outre des considérations pratiques pour concilier le bénéfice des ordres et le coût de leur calcul.

Cet inventaire met en évidence la diversité des cas d'utilisation des ordonnancements de nœuds tout en montrant des tendances communes dans l'intention sous-jacente : placer les nœuds côte à côte s'ils sont connectés ensemble ou avec les mêmes nœuds, s'assurer que les nœuds successifs dans l'ordre sont connectés dans le graphe, classer les nœuds selon une notion d'importance, représenter les propriétés de connectivité pour les groupes de nœuds, ou appliquer une contrainte sur l'emplacement relatif des nœuds dans l'ordre. Nous identifions quatre méthodes communes utilisées par la plupart des algorithmes pour calculer les ordres : l'optimisation d'une fonction mathématique, la sélection des nœuds selon une notion de priorité, la réduction du graphe pour analyser un graphe plus petit, ou le traitement successif des nœuds pour décider de leur emplacement dans l'ordre.

Chapitre 4
Ordres pour accélérer les calculs
Réduire le taux de défaut de cache des algorithmes de graphe

Ce chapitre développe l'application des ordres de nœuds au problème de l'optimisation du cache. La contribution est une réplication complète d'un article influent qui compare divers ordres sur une sélection d'algorithmes. Nous fournissons également des discussions supplémentaires sur les choix algorithmiques et sur la performance des différents ordres. Le projet inclut une implémentation open-source de tous les ordres et algorithmes de l'article, ce qui manquait auparavant dans la littérature.

Les personnes ayant écrit l'article initial conçoivent une fonction mathématique qui estime l'efficacité d'un système de cache pour un algorithme de graphe arbitraire, et elles créent un nouvel ordre appelé Gorder afin d'optimiser cette fonction. En testant leur ordre pour une variété d'algorithmes de graphes, elles affirment que Gorder conduit à une accélération moyenne de 20% par rapport à lorsque l'ordre initial du graphe est conservé. Nos résultats confirment cette tendance et montrent que les performances de Gorder sont dues à une meilleure utilisation du cache. Cependant, nous nuançons ce résultat en montrant que d'autres ordres comme le parcours en largeur sont presque aussi performants, alors qu'ils reposent sur des concepts algorithmiques plus simples, ont une complexité temporelle plus faible et peuvent être calculés beaucoup plus rapidement. Le temps de calcul de Gorder n'évolue pas linéairement quand les graphes grandissent, ce qui le rend impraticable pour les grands graphes et souligne le compromis à trouver entre le temps de calcul de Gorder et l'accélération substantielle qu'il apporte aux algorithmes de graphes effectués ensuite. Par ailleurs, nous analysons un paramètre impliqué dans le calcul de Gorder et montrons qu'il n'a pas autant d'impact sur la qualité de l'ordre que ce que l'on pensait jusqu'à présent.

Chapitre 5
Ordres pour réduire le nombre d'opérations
Énumérer les triangles d'un graphe non-orienté

Ce chapitre aborde une question importante de l'exploration des données : l'énumération des triangles d'un graphe. La principale contribution est un ensemble de nouveaux ordres qui accélèrent de manière significative les algorithmes d'énumération de triangles existants. Dans le contexte des grands graphes réels, les algorithmes actuels les plus rapides emploient des ordres de nœuds. Tout d'abord, les nœuds reçoivent des indices dans un ordre spécifique. Ensuite, les arêtes sont dirigées des indices de nœuds inférieurs vers les indices supérieurs : elles pointent toutes dans une direction spécifique. Au lieu de balayer les arêtes jusqu'à ce que trois d'entre elles forment un triangle, l'algorithme suit la direction des arêtes et les triangles sont identifiés plus rapidement. Pour simplifier, l'algorithme explore des chemins de longueur deux : d'un nœud d'origine, en passant par un nœud intermédiaire, jusqu'à un nœud de destination ; si l'origine et la destination sont reliées par une troisième arête, un triangle a été trouvé. Le temps d'exécution de l'algorithme dépend du nombre de chemins de longueur deux ; mais rappelons que les chemins sont définis par la direction des arêtes, qui sont données par les indices des nœuds. Le choix de l'ordre aura donc un impact sur le temps que l'algorithme met à trouver tous les triangles.

Si les algorithmes de listage de triangles de la littérature utilisent cette technique pour accélérer leur exécution, ils utilisent des ordres de base qui offrent des garanties mathématiques générales. Au contraire, notre travail étudie une fonction mathématique qui représente le nombre d'opérations des algorithmes de listage de triangles standard en fonction de l'ordre des nœuds. Après avoir prouvé que la recherche d'un ordre optimal est computationnellement difficile, nous proposons différentes façons d'obtenir une solution approximative, certaines se concentrant sur la qualité de l'ordre et d'autres sur la vitesse de son calcul.

Ces nouveaux ordres comblent un vide dans la littérature : bien que deux algorithmes avec des formules de complexité différentes aient été découverts, les ordres de nœuds n'ont été conçus que pour le premier. Nos ordres sont spécifiquement adaptés au second, ce qui permet d'accélérer le calcul. Nos expériences sur de grands réseaux réels confirment que les nouveaux ordres accélèrent de manière significative l'énumération des triangles.

Chapitre 6
Ordres pour certifier la qualité
Certification de qualité pour des heuristics de couverture par sommets

Ce chapitre aborde sur les réseaux réels le problème computationnellement difficile de la recherche d'une couverture minimale par sommets. La contribution est une méthode pour certifier la qualité d'une solution approximative lorsque la solution optimale n'est pas disponible ou trop coûteuse à calculer. Bien que tous les algorithmes connus pour résoudre le problème de la couverture par sommets aient une complexité exponentielle dans le pire des cas, nous exploitons deux algorithmes qui s'exécutent en temps linéaire : l'un trouve une solution approximative pour la couverture par sommets, et l'autre calcule une limite inférieure de la solution optimale. Ensemble, ces deux résultats certifient que la couverture approximative est proche de la solution optimale. Pour ce faire, nous exploitons des ordres de nœuds simples basés sur le degré des nœuds. Les algorithmes de bornes sont basés sur d'autres problèmes de graphes importants, à savoir les problèmes

de couplage maximum et de partition en cliques.

Nous effectuons des expériences approfondies sur un grand nombre de réseaux réels ; la méthode de certification qui utilise les couvertures par sommets garantit que les couvertures par sommets approximatives sont presque optimales sur tous les graphes. Cela indique que les réseaux réels possèdent des propriétés qui facilitent le problème de la couverture par sommets. Le principe de la méthode de certification est général et nous en esquissons des extensions à quelques autres problèmes de graphes.

Chapitre 7
Science des réseaux
Profils de mobilité dans l'espace des connaissances scientifiques

Ce chapitre présente un travail conjoint avec une équipe de science des données et qui ne concerne pas les ordres de nœuds et dépasse donc la ligne directrice de la thèse. Nous étudions le réseau de collaboration donné par une collection d'articles scientifiques. La contribution est une analyse des trajectoires des scientifiques dans l'espace de la connaissance au cours de leur carrière, qui montre des schémas communs avec les déplacements humains sur le globe. Nous utilisons toutes les publications d'arXiv [1] pour créer un espace à haute dimension qui représente les différents domaines scientifiques et les articles correspondants. Nous dessinons la trajectoire des scientifiques, c'est-à-dire la séquence des domaines dans lesquels ils ont publié leurs articles. En projetant cet espace en deux dimensions, nous pouvons le comparer à la mobilité géographique.

Les résultats statistiques montrent que les flux de trajectoires entre deux lieux de connaissance sont compatibles avec un modèle de gravité : les scientifiques peuvent s'éloigner de leur domaine initial, mais seulement si le domaine de destination est très actif. Nous identifions deux types de profils de mobilité individuels entre lesquels nous pouvons séparer les peronnes qui font de la recherche : les exploratrices, qui s'engagent dans la recherche interdisciplinaire et ouvrent de nouveaux domaines, et les exploitantes, qui restent dans leur domaine et développent une expertise.

Perspectives

Les contributions de cette thèse et la vaste littérature antérieure montrent que les ordres de nœuds sont un élément crucial des algorithmes de graphes. Pour cette raison, nous pensons qu'une analyse plus systématique des ordres pourrait conduire à de nouveaux résultats. Tout d'abord, lorsque nous soupçonnons qu'un algorithme ou une technique est influencé par l'ordre des nœuds, nous pourrions tester systématiquement les ordres existants afin de mieux comprendre quels sont les mécanismes qui fonctionnent le mieux. La classification du chapitre 3 pourrait contribuer à cet objectif, ainsi que les différents ordres que nos implémentations proposent (voir Introduction). Deuxièmement, les ordres définis par des fonctions objectives pourraient bénéficier d'une analyse théorique plus approfondie : une façon classique de les traiter, que nous avons suivie dans le chapitre 5, est de prouver leur difficulté, puis de concevoir des heuristiques dont on pense intuitivement qu'elles améliorent la valeur de la fonction. Bien que les résultats expérimentaux soient un moyen valable de prouver le succès de ces heuristiques, une base mathématique plus solide du problème difficile initial est également importante. Dans cette direction, la conception d'algorithmes ou de schémas d'approximation est intéressante, de même que la recherche de classes de graphes ou de paramètres pour lesquels

[1] arXiv est une plateforme où les scientifiques partagent leurs articles en libre accès : https://arxiv.org.

le problème est abordable. La recherche de bornes pour l'optimisation est également une piste intéressante à explorer, comme dans le chapitre 6 : les bornes révèlent à quel point l'heuristique est proche de l'optimum, ce qui indique s'il vaut la peine de déployer des efforts supplémentaires pour s'approcher de l'optimum.

Dans certains problèmes de graphes, l'ordre des arêtes peut être plus pertinent que l'ordre des nœuds. Notez qu'il existe des moyens naturels de définir l'un à partir de l'autre : étant donné un ordre de nœuds, on peut ordonner les arêtes selon le plus petit indice de leurs nœuds ; inversement, étant donné un ordre d'arêtes, on peut ordonner les nœuds selon l'indice de la première arête dans laquelle ils apparaissent. Dans cette thèse, nous nous concentrons sur l'ordre des nœuds en nous appuyant sur deux hypothèses : le graphe est stocké sous forme de listes d'adjacence et la principale opération atomique des algorithmes consiste à énumérer les voisins d'un nœud. Cependant, l'ordre des arêtes s'avère pertinent dans d'autres contextes, lorsque les arêtes sont l'élément clef au lieu des nœuds. La première hypothèse est invalidée lorsque le graphe est stocké sous forme de liste d'arêtes, ce qui est un format standard pour partager des ensembles de données dans un fichier. La deuxième hypothèse est inexacte pour certains algorithmes basés sur les arêtes, notamment l'algorithme gourmand de couplage décrit dans le chapitre 6. De même, certaines méthodes de calcul du Pagerank et des centralités par vecteurs propres reposent sur l'énumération des arêtes, et l'ordre des arêtes peut donc influer sur la vitesse de convergence de ces algorithmes. Dans une matrice d'adjacence, chaque élément correspond à une arête ; l'énumération des arêtes suivant l'ordre des nœuds correspondants revient à lire la matrice ligne après ligne. Cependant, il ne s'agit là que d'une des nombreuses possibilités d'énumérer les arêtes, et d'autres ordres ont été proposés. L'un d'entre eux est l'ordre des courbes bidimensionnelles de remplissage de l'espace : le tracé d'une courbe qui traverse tous les éléments de la matrice d'adjacence fournit un ordre des arêtes qui ne correspond pas à l'ordre des nœuds. Des recherches sur les ordres des arêtes nous aideraient à comprendre comment les ordres des nœuds et des arêtes sont corrélés, et quel type d'ordre est le mieux adapté à un problème de graphe particulier.

Les ordres constituent un moyen caché de rendre les algorithmes plus rapides ou plus efficaces et, à ce titre, ils pourraient être intégrés dans des systèmes de traitement des graphes. Ces systèmes logiciels permettent généralement aux personnes utilisatrices de stocker, de modifier et de consulter efficacement de grands graphes de manière parallèle ou distribuée. Pour ce faire, ces systèmes nécessitent des algorithmes et des structures de données spécialisés pour effectuer des opérations telles que la traversée de graphes, la fouille de motifs ou la détection de communautés. Ordonner les nœuds d'une manière spécifique pourrait aider à accélérer ces tâches en général, comme on l'a vu avec l'optimisation du cache dans le chapitre 4. Il est même possible de maintenir plusieurs ordres de manière transparente, de sorte que chaque type de requête puisse être traité avec un ordre approprié. Par exemple, si le système maintient un ordre des nœuds basé sur leur degré, il peut traiter plus efficacement les requêtes qui reposent sur le degré, comme l'énumération des triangles dans le chapitre 5 ou d'autres tâches de fouille de motifs. Simultanément, le système peut maintenir un ordre des nœuds avec un mécanisme de localité pour améliorer les résultats des algorithmes qui tirent parti de cette propriété, comme le partitionnement des graphes. Étant donné le nombre limité de mécanismes d'ordonnancement impliqués dans les domaines d'application, les systèmes de traitement des graphes pourraient faire face à une diversité de requêtes avec seulement quelques ordres à maintenir, ce qui rendrait le surcoût de temps et d'espace raisonnable.

En outre, il est possible d'appliquer les techniques d'ordre à des modèles de graphes plus sophistiqués, notamment les graphes temporels ou dynamiques. Pour un graphe

qui évolue dans le temps du fait de l'ajout ou de la suppression de nœuds et d'arêtes, le défi consiste à maintenir un ordre exact ou approximatif avec des propriétés spécifiques. Si maintenir l'ordre des degrés est possible avec une file de priorité, ce n'est pas évident pour des ordres plus élaborés tels que l'ordre par dégénérescence ou Slashburn. En effet, ils sont basés sur une décomposition du graphe qui peut changer sensiblement par l'ajout d'une seule arête. Une extension du problème de la maintenance consisterait à concevoir des ordres robustes aux modifications du graphe. De tels ordres seraient les meilleurs candidats pour résoudre le compromis du temps de calcul : on peut accepter de passer plus de temps à calculer un ordre si l'on a la garantie qu'il peut être mis à jour en peu de temps lorsque le graphe change.

En outre, il est possible d'appliquer les techniques d'ordre à des modèles de graphes plus sophistiqués, notamment les graphes temporels ou dynamiques. Pour un graphe qui évolue dans le temps du fait de l'ajout ou de la suppression de nœuds et d'arêtes, le défi consiste à maintenir un ordre exact ou approximatif avec des propriétés spécifiques. Si maintenir l'ordre des degrés est possible avec une file de priorité, ce n'est pas évident pour des ordres plus élaborés tels que l'ordre par dégénérescence ou Slashburn. En effet, ils sont basés sur une décomposition du graphe qui peut changer sensiblement par l'ajout d'une seule arête. Une extension du problème de la maintenance consisterait à concevoir des ordres robustes aux modifications du graphe. De tels ordres seraient les meilleurs candidats pour résoudre le compromis du temps de calcul : on peut accepter de passer plus de temps à calculer un ordre si l'on a la garantie qu'il peut être mis à jour en peu de temps lorsque le graphe change.

Enfin, les ordres ne se limitent pas aux problèmes de graphes et peuvent également être liés au problème algorithmique du plongement. Un plongement peut être considéré comme une centralité multidimensionnelle qui recherche un placement optimal des éléments pour satisfaire certaines contraintes, telles que le regroupement d'éléments ou le maintien de la variabilité sur un petit nombre de dimensions. En explorant l'utilisation des ordres dans ce problème qui ne concerne pas les graphes, nous pourrions découvrir de nouvelles idées et techniques qui s'appliquent aux problèmes de graphes. Par exemple, le concept de localité dans les ordres de nœuds peut être dérivé d'un plongement de nœuds, comme suggéré dans le chapitre 7. Ainsi, les ordres peuvent servir d'outil puissant pour traiter un spectre plus large de problèmes algorithmiques.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

The aim of this thesis is to use theoretical tools of computer science to improve algorithms in practice. An algorithm is a set of instructions to follow in order to answer a question starting from some input data. Algorithms are usually designed for computers, but humans use them too. Consider as an example the primary school method to add two large numbers: the teacher said, add up the rightmost digits, write down the units of the result, and report the carry if necessary; then repeat with the next digit. As an illustration for readers who are not familiar with algorithms, Algorithm 1, below, shows how such a procedure can be formalised: it starts with two numbers x and y given by their digits, and a resulting sum z that is initially zero (line 1). Reading the indices i from right to left (line 2), it adds the i-th digits of x and y with the possible carry z_i (line 3). The unit digit of the result w corresponds to the i-th digit of the result (line 5), while the tens of w carry to the next iteration of the loop (line 6). Once all digits have been treated, z contains the result of the sum $x + y$.

Algorithm 1 – Manually adding two numbers

Input: x and y two integers given in their decimal writing $x_a x_{a-1} \dots x_1$ and $y_a y_{a-1} \dots y_1$.
Output: $z = x + y$ given in its decimal writing $z_{a+1} z_a \dots z_1$.

1: define z by its digits $z_1 = z_2 = \dots = z_{a+1} = 0$
2: **for** i going from 1 to a **do**
3: compute $w = x_i + y_i + z_i$ ▷ x_i, y_i, z_i are digits between 0 and 9 so $w \leq 27$
4: write the two digits of $w = w_2 w_1$ ▷ w_2 may be zero
5: set $z_i = w_1$ ▷ final result for digit i
6: set $z_{i+1} = w_2$ ▷ carry for the next step
return $z = z_{a+1} z_a \dots z_1$

The work presented in this manuscript focuses on algorithms that process data in the form of *graphs* rather than integers like Algorithm 1. A graph is a mathematical object that represents interacting elements; an element is called a *node* or a *vertex*, and two elements that interact are linked by an *edge*. An example of graph is displayed in Figure 1.1. Computer scientists have long been interested in the properties of these objects, and they have designed theoretical algorithms for arbitrary graphs: checking whether the whole graph is connected, finding the shortest paths or the largest set of inter-connected nodes, etc. They have also studied models of random graphs where the nodes have a certain probability of interacting. However, graphs are also a simple structure to describe a number of real-world situations: international trade, traffic jams, scientific collaboration... Experts of the corresponding fields can collect this data and share it to others as a graph file. Studying these *real-world networks* differs from studying mathematical models of graphs: they have specific properties due to the situation from which they arise, but these properties are not known in advance.

The main challenge associated with real-world networks is their gigantism. Our collective ability to gather and organise data makes increasingly large graphs available: online social networks can track millions of users and their billions of interactions, trillions of emails are sent every year between billions of internet users, and biologists are actively working on mapping the 10^{14} (hundred thousand billion) synaptic connections of a human brain.

How long will an algorithm take to process such graphs? In other words, is it possible to design algorithms that *scale* to the largest real-world networks? The relation between the size of the graph and the execution time of the algorithm is called the *time complexity*. For large graphs, the aim is to have a linear complexity, or a time roughly proportional to

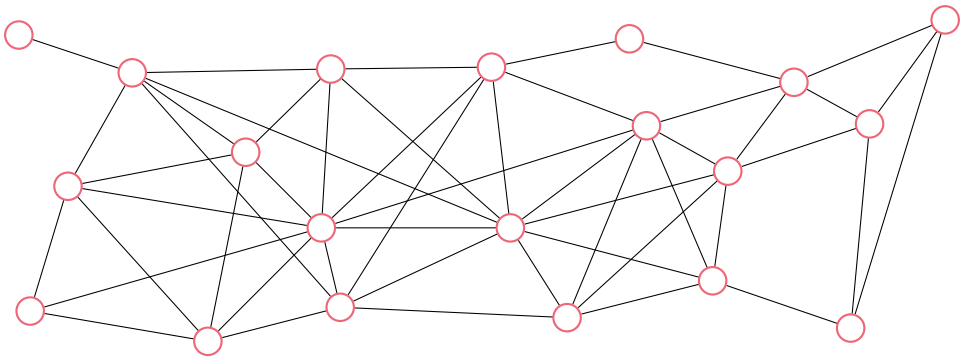


Figure 1.1: **Example of graph** with 20 nodes (red) and 50 edges (black).

the size of the graph, where the size is the number of nodes or edges of the graph. For some standard questions, unfortunately, the scientific community only knows algorithms that answer in exponential time in the worst case. The worst-case scenario is often studied in algorithmics to qualify the difficulty of a problem. Yet, as mentioned earlier, real-world networks have specific properties that distinguish them from the worst case, so much that many algorithms with high worst-case complexity have been observed to behave quasi-linearly on real-world networks. Hence the general scalability constraints for this thesis: in terms of complexity, algorithms should have a reasonable (not exponential) theoretical complexity on the considered classes of graphs; in terms of execution time, they should scale quasi-linearly and take less than a few hours to process a network with billions of edges on a standard laptop (as long as it has enough memory).

To address the scalability issue, this thesis proposes to use *node orderings*. Sorting the nodes of a graph in a relevant order has proved to be key for solving various problems in large graphs. To convince ourselves that orderings matter, let us take the example of additions again: if we need to mentally calculate 19+7+3, in which order will we do it? Starting with 19+7 is not straightforward, while we may recognise 7+3=10 directly, after which adding 19 is simpler. In both cases, the result is 29 and we manage to find it. Yet, the order in which we process the elements can change the total time of the algorithm. The same phenomenon arises in graph algorithms on a computer: choosing an appropriate node ordering can make the execution faster or the result more accurate. Hence the question that will guide us throughout this manuscript:

How can node orderings help us
design scalable algorithms
for real-world networks?

To answer this question, a crucial step is to identify which algorithmic graph problems can benefit from node orderings. Graphs normally represent interacting elements in an unordered way: there is no natural concept of first node in a graph. For this reason, most graph algorithms do not seem to involve node orderings at first sight. On the other hand, the fact that algorithms handle pieces of data one by one lets an implicit ordering emerge: there is a first node for the algorithm. The challenge is to understand for which algorithms this ordering matters and what gains a reordering may bring: time, quality, mathematical guarantee...

Once a suitable problem has been identified, the main task consists in designing an ordering that enhances the corresponding algorithms. Fortunately, there is a large repertoire of orderings that can be called upon, many of which were designed with scalability constraints in mind. Among them, the most versatile is perhaps the degree ordering: nodes

are ranked according to the number of edges in which they are involved. New problems may reuse such an ordering or adapt it to their distinctive characteristics. It is also common to define new orderings, in which case a particular attention is paid to scalability issues when it comes to practical applications.

To validate the impact of an ordering, both mathematical proofs and experimental evidence are critical. Mathematical proofs are a way to ground the ordering method into existing theory and to guarantee the behaviour of algorithms in any circumstance: for instance, proving that a specific ordering reduces the time complexity of an algorithm gives a new insight on the problem, which can lead to further theoretical results. Experimental evidence emphasises use cases where the ordering method has a quantitative impact and points towards possible real-world applications.

Considering these three tasks of problem identification, ordering design, and validation, here is the organisation of the manuscript and our main contributions:

Chapter 2 introduces the computer science concepts and notations used throughout the thesis. It first presents a background on graph theory with notions of algorithmic complexity and data structures. It then outlines the challenges of graph mining associated with real-world networks and it presents the sources and properties of the datasets that are used in the experiments. It also discusses the hardware issues related with fluctuations in time measurements.

Chapter 3 reviews existing node ordering methods across a panel of application domains. The contribution is an extensive list of orderings that appear under different names and shapes in the literature, as well as a classification of the main mechanisms that call for orderings and algorithmic methods used to compute them. Additionally, we discuss cases where orderings can represent other problems of graph theory, and practical considerations to balance the gain of orderings with the cost of computing them.

This review highlights the diversity of use cases for node orderings, while showing common patterns in the underlying intention: placing nodes together if they are connected together or with the same nodes, ensuring that successive nodes in the ordering are connected in the graph, ranking the nodes according to a notion of importance, representing connectivity properties for groups of nodes, or enforcing a constraint on the relative location of nodes in the ordering. We identify four common methods used by most of the algorithms to compute orderings: optimising a mathematical function, selecting nodes according to a notion of priority, reducing the graph to analyse a smaller graph, or streaming the nodes to decide where to place them in the ordering.

Chapter 4 develops the application of node orderings to the problem of cache optimisation. The contribution is a full replication of an influential paper that compares various orderings over a selection of algorithms. We also provide additional discussions on the algorithmic choices and on the performance of different orderings. The project includes an open-source implementation of all the orderings and algorithms of the paper, which was previously missing from the literature.

The authors of the initial paper design a mathematical function that estimates the efficiency of a cache system for an arbitrary graph algorithm, and they create a new ordering called Gorder to optimise this function. Testing their ordering for a variety of graph algorithms, they claim that Gorder leads to an average speedup of 20% compared to keeping the initial ordering of the network. Our results confirm this tendency and give evidence that the performance of Gorder is due to a better use of the cache. Yet, we mitigate

the result by showing that other orderings perform almost as well although they rely on simpler algorithmic concepts, have a lower time complexity, and can be computed much faster. Additionally, we analyse a parameter involved in computing Gorder and show that it does not impact the quality of the ordering as much as what was previously thought.

Chapter 5 addresses an important question of pattern mining: enumerating the triangles of a graph. The main contribution is a set of new orderings that significantly accelerate state-of-the-art triangle listing algorithms. We study a mathematical function that represents the number of operations of standard triangle listing algorithms depending on the node ordering. After proving that finding an optimal ordering is computationally hard, we propose different ways to obtain an approximate solution, some that focus on the quality of the ordering, and some that focus on the speed of its computation.

These new orderings bridge a gap in the literature: although two algorithms with different complexity formulas have been discovered, node orderings have only been designed for the first one. Our orderings are specifically tailored for the second one, which leads to a speedup that we assess in experiments with a panel of large real-world graphs.

Chapter 6 tackles on real-world networks the computationally hard problem of finding a minimum vertex cover. The contribution is a method to certify the quality of an approximate solution when the optimal solution is unavailable or too costly to compute. Although all the algorithms known to solve the vertex cover problem have an exponential worst-case complexity, we leverage two algorithms that run in linear time: one that finds an approximate solution for vertex cover, and one that computes a lower-bound on the optimal solution. Together, these two results certify that the approximate cover is close to the optimum. To do so, we exploit simple node orderings based on the degree of nodes. The bounding algorithms are based on other important graph problems, namely the maximum matching and the clique cover problems.

We perform thorough experiments on a large panel of real-world networks; the certification method that uses clique covers is able to guarantee that approximate vertex covers are almost optimal on all the graphs. This indicates that real-world networks have properties that make the vertex cover problem easier. The principle of the certification method is general and we sketch extensions of it to a few other graph problems.

Chapter 7 presents a joint work with a data science team that does not involve node orderings and is thus beyond the main scope of the thesis. We study the collaboration network given by a collection of scientific papers. The contribution is an analysis of the trajectories of researchers in knowledge space across their career, which shows common patterns with the movements of humans on the globe. We use all the publications of arXiv [1] to create a high-dimensional space that represents the different scientific fields and the corresponding papers. We draw the trajectory of researchers, which means the sequence of fields in which they published their papers. Projecting this space in two dimensions, we are able to compare it with geographic mobility.

Statistical results show that the flows of trajectories between two knowledge locations comply with a gravity model: researchers may jump far away from their initial field, but only if the destination field is very active. We identify two types of researchers according to their individual mobility patterns: explorers, who engage in interdisciplinary research and open new fields, and exploiters, who stay within their field and develop an expertise.

[1] arXiv is a platform where researchers share their papers in open-access: https://arxiv.org.

Publications

* Chapter 4: [*Replication*] *Speedup graph processing by graph ordering*
  Lécuyer, Danisch, and Tabourier [2021], ReScience.
* Chapter 5: *Tailored vertex ordering for faster triangle listing in large graphs*
  Lécuyer, Jachiet, Magnien, and Tabourier [2023a], ALENEX.
* Chapter 6: *Vertex cover quality certification on real-world networks*
  Lécuyer, Tabourier, and Magnien [2023b], submitted ESA.
* Chapter 7: *Charting mobility patterns in the scientific knowledge landscape*
  Singh, Tupikina, Lécuyer, Starnini, and Santolini [2023], submitted EPJ DataScience.

Talks

* June 2022, FRCCS conference, France: Chapter 5.
* August 2022, MLG workshop of KDD conference, USA: Chapter 5.
* November 2022, JGA workshop, France: Chapter 6.
* January 2023, ALENEX conference, Italy: Chapter 5.
* January 2023, seminar at University of Milan, Italy: Chapters 5 and 6.
* April 2023, seminar at University of British Columbia, Canada: Chapters 3, 5 and 6.
* May 2023, FRCCS conference, France: Chapter 6.

Open-source repositories

* Chapter 4: https://github.com/lecfab/rescience-gorder
* Chapter 5: https://github.com/lecfab/volt
* Chapter 6: https://github.com/lecfab/certifVC

Chapter 2

Preliminaries

Graph mining and network analysis

In this chapter, we introduce the mathematical concepts and notations that will be used throughout the manuscript. We begin by presenting key tools of graph theory, including algorithmic complexity and graph data structures. We then expose the specifics of graph mining problems when dealing with large real-world networks. Finally, we discuss the hardware issues that can be faced while trying to accurately measure the execution time of algorithms.

2.1 Graph theory and algorithmics

Graph theory is a branch of mathematics and computer science that studies objects made of relations between elements. While these discrete objects have been studied for centuries as pure mathematical constructions, they were originally devised to represent real-world situations, which calls for applied methods and efficient algorithms.

2.1.1 Definitions and notations

Formally, a *graph* G is a pair (V, E), where V is a set of n elements called the *nodes* or *vertices* of the graph, and E is a set of m elements of $V \times V$ called the *edges* or *links* of the graph. In an *undirected graph*, an edge $e \in E$ can be seen as a set of two distinct nodes ($e = \{u, v\}$) or two symmetric pairs ((u, v) and (v, u)). We say that u and v are adjacent, that e is incident to u and v, and that v is a neighbour of u (and conversely). The set of neighbours of a vertex u is denoted $N_u = \{v, \{u, v\} \in E\}$, and its degree is the number of neighbours $d_u = |N_u|$. In a *directed graph*, an edge or *arc* $e \in E$ is a directed pair of distinct nodes: $e = (u, v)$. We say that e connects u to v, that u is a predecessor of v and v is a successor of u. The set N_u of neighbours of u is partitioned into its predecessors N_u^- and successors N_u^+, which also defines the indegree $d_u^- = |N_u^-|$ and the outdegree $d_u^+ = |N_u^+|$; their sum is $d_u^- + d_u^+ = d_u$.

A *subgraph* of G is a graph $G' = (V', E')$ made of a subset of nodes from the original graph along with some of the edges that connect them. More precisely, V' is a subset of V and E' is a subset of E that only contains edges with both endpoints in V': $V' \subseteq V$ and $E' \subseteq E \cap V' \times V'$. If $E' = E \cap V' \times V'$, then G' is an *induced* subgraph. Some subgraphs are of particular interest in graph theory. A *walk* is a sequence of edges such that an edge finishes on the node where the next edge in the sequence begins. A *path* is a walk where all the nodes are distinct. A *cycle* is a walk of distinct edges (at least three) that starts and ends at the same node. A *triangle* is a set of vertices $\{u, v, w\}$ such that $\{u, v\}, \{v, w\}, \{u, w\} \in E$ (for an undirected graph). A *k-clique* is a set of k vertices that are all fully connected, meaning that there is an edge between every pair of vertices in the set; a triangle is thus a 3-clique.

Graph theory has specific methods and algorithms for some particular types of graphs. A graph is considered *connected* if there exists a path between any two nodes, otherwise it can be broken down into maximal connected subgraphs known as *connected components*. A *tree* is a connected graph that contains no cycles, meaning that there is only one path between any two nodes. A *bipartite* graph is a graph that has no cycle of odd length; its nodes can be partitioned into two sets such that every edge connects a node of the first set with a node of the second set, and there is no edge within a set. A graph is *planar* if it can be drawn on a sheet of paper without any of its edges crossing.

In addition to these definitions, other attributes can be added to the structure of the graph. In a *weighted* graph, a function $w : E \to \mathbb{R}$ gives a numerical value to the edges to represent their length or capacity. A *labelled* graph assigns a category or a value to its

nodes in order to distinguish several types of nodes; node orderings are a type of node labelling. Some graph models include *loops*, which are edges that connect a node to itself. A *multigraph* is a model of graph where the same pair of nodes can be connected by multiple edges, in which case they may have distinct labels and represent different types of links. Describing more complex relations sometimes require *hypergraphs*, which allow edges to connect more than two nodes: an edge is a set of nodes with possibly more than two elements.

2.1.2 Algorithmic complexity

In the field of algorithmics, the question of computational complexity consists in classifying problems and algorithms based on the resources required to solve them, namely time and space. This classification involves two distinct aspects: the complexity of an algorithm, which estimates the resources that it requires, and the complexity of a problem, which represents the minimum resources necessary for an algorithm to solve it.

Time complexity of algorithms

The time complexity of an algorithm is a measure of the amount of time that an algorithm takes depending on the size of the input. It represents an estimate of the number of operations that an algorithm will perform, and is usually expressed as a function of the input size. For graph algorithms, the size is mainly represented by the number of nodes n and edges m, but it can sometimes be parametrised by the properties of the graph such as the largest degree or the number of triangles.

Time complexity is often used to compare the efficiency of different algorithms that solve the same problem, and to determine whether a given algorithm can solve large-scale problems. It is not an exact count of the number of operations that the algorithm makes, rather a description of the evolution of this number when the size of the input increases. To describe this evolution, we use the following asymptotic notations. For two functions f and g, we write $f = \mathcal{O}(g)$ when there exist a factor $k \in \mathbb{R}^+$ and a threshold x_0 such that, for all $x \geq x_0$, $f(x) \leq k \cdot g(x)$. We say that f is asymptotically bounded by g: for large enough values of x, the function f is lower than g up to a constant factor. When we have both $f = \mathcal{O}(g)$ and $g = \mathcal{O}(f)$, we write $f = \Theta(g)$: functions f and g are asymptotically within a constant factor of each other. If this factor can be arbitrarily close to one, we say that f and g are equivalent and write $f \sim g$.

In the context of the complexity of graph algorithms, the function f represents the number of operations that the algorithm needs. As for the function g, we will usually take a simple mathematical function as a reference, for instance the square function $x \mapsto x^2$, the logarithm $x \mapsto \log(x)$, etc. If we take the number of edges m as the size of the graph, we will say that an algorithm has linear complexity when its number of operations is in $\mathcal{O}(m)$, quadratic for $\mathcal{O}(m^2)$, exponential for $\mathcal{O}(2^m)$, logarithmic for $\mathcal{O}(\log m)$, and quasi-linear for $\mathcal{O}(m \log m)$. A polynomial complexity means that there is an exponent $\alpha \in \mathbb{R}^+$ such that the algorithm needs $\mathcal{O}(m^\alpha)$ operations. A constant complexity $\mathcal{O}(1)$ or $\Theta(1)$ means that the number of operations does not depend on the input size.

Let us stress that the time complexity does not necessarily reflect the time that an algorithm takes. First, recall that the notation ignores multiplicative factors and dominated terms, so that algorithms with m or $5m + 2^{100}$ operations are both linear, and algorithms with $\log_2(m)$, $\log_{10}(m)$ or $\ln m$ operations are all logarithmic. However, these factors and terms influence the execution time of the algorithm.

Second, the $\mathcal{O}$ notation represents a worst-case upper-bound. Take for instance an algorithm that is cubic in the number of nodes, which corresponds to a complexity $\mathcal{O}(n^3)$. This notation means that the algorithm can process *any* graph of n nodes in *at most* $\mathcal{O}(n^3)$ operations: there may be some graphs that require less operations, and there may be a lower valid estimate of the complexity. To give a more precise indication of the scalability, one possibility is to study the average-case complexity. Another option is to use the Θ notation, which means that the algorithm needs this asymptotic number of operations for any input. As we will see, the worst-case analysis partly explains why algorithms can perform well on real-world graphs even though their time complexity is high: real-world is not the worst.

Apart from time complexity, the question of space complexity of algorithms is also important: it represents the memory required by an algorithm to process an input of a certain size. The same asymptotic notations can be used, with the same caveats. Importantly, the space complexity is always lower than the time complexity, as filling the memory requires time.

Computational complexity of problems

The complexity of an algorithmic problem can be seen as the lowest time complexity among all the algorithms that solve the problem. The field of computational complexity theory provides a detailed hierarchy of complexity for algorithmic problems, consisting of hundreds of different classes [1]. However, for the purpose of this thesis, we will focus on a specific part of this hierarchy, namely the distinction between problems in P and NP. These categories concern decision problems, which correspond to yes-no questions.

A decision problem is in P when there exists an algorithm with polynomial time complexity that solves it for any instance. The exponent of this polynomial, or the multiplicative factors attached to it, can be arbitrarily high, which does not indicate that such an algorithm can solve big instances in practice. Facing a P problem, we will be interested in finding algorithms with low exponents, aiming at the ultimate goal of designing a linear or quasi-linear algorithm.

A decision problem is in NP when its solution can be verified in polynomial time, which includes problems in P. The open question of *P versus NP* stresses that it is still unknown whether all the problems of NP are also in P; at the moment, many NP problems have no known polynomial algorithm. Some NP problems have been shown to be as hard as any other NP problem, in the sense that solving them translates into a solution for the other problems; they are called NP-complete. Facing a NP problem that is not known to be in P, we will be interested in proving that it is NP-complete, and in finding subclasses of instances for which the problem is in P. We are also interested in the FPT class (fixed-parameter tractability) that contains the problems that become polynomial when a given parameter of the instances is a constant.

This classification applies on decision problems, which are yes-no questions. When it comes to optimisation problems such as finding the minimum of a function, it is common to obtain problems that are possibly harder than NP-complete problems, and we call them NP-hard. Facing a NP-hard optimisation problem, we will seek algorithms of lower complexity that solve it approximately [2].

Whenever the complexity of an optimisation problem is too high for practical applications, it is interesting to design a constant-factor approximation. For a minimisation prob-

[1] For a list of complexity classes and their interconnections, see https://complexityzoo.net

[2] For a list of approximation results, see https://www.csc.kth.se/~viggo/wwwcompendium/wwwcompendium.html

Data structure	Space complexity	Time complexity of operations		
		Adjacency checking	Neighborhood listing	Edge listing
Adjacency matrix	$\Theta(n^2)$	$\mathbf{\Theta(1)}$	$\Theta(n)$	$\Theta(n^2)$
Edge list	$\mathbf{\Theta(m)}$	$\mathcal{O}(m)$	$\Theta(m)$	$\mathbf{\Theta(m)}$
Adjacency lists	$\Theta(n+m)$	$\mathcal{O}(d_u)$	$\mathbf{\Theta(d_u)}$	$\Theta(n+m)$

Table 2.1: **Graph data structures and their associated complexities.** Bold font represents the lowest complexity of a given column. In connected graphs, $n \leq m$ so $\Theta(n+m) = \Theta(m)$, which makes the edge list and adjacency lists equivalent for space complexity and edge listing.

lem with minimum value x, an algorithm is said to be a k-approximation if it outputs a solution of value y with the guarantee that $x \leq y \leq kx$. In other words, its solution cannot be too far off the minimum [3]. A problem that has a polynomial-time constant-factor approximation is called approximable, and the class of these problems is noted APX. If there exists a polynomial k-approximation for any $k > 1$, we say that the problem has an approximation scheme and belongs to the class PTAS.

2.1.3 Data structures

This section presents data structures that are commonly used in graph algorithms. To represent a graph itself, several data structures can be used in practice. The choice of structure depends on the size of the graph, the available memory, the mathematical representation of an algorithm, or the type of operations that one chooses to favour. We will consider three graph operations to compare the pros and cons of the different data structures: adjacency checking, which consists in finding whether two nodes are adjacent, neighbourhood listing, where all the neighbours of a given node are enumerated, and edge listing, where all the edges of the graph are enumerated. Table 2.1 sums up the time and space complexities of the three data structures that we describe hereafter.

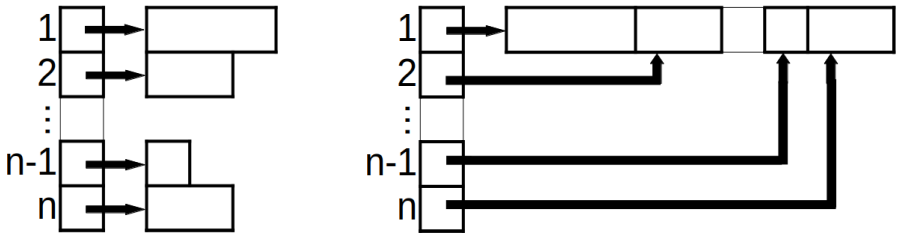
The most natural structure to describe a graph is perhaps the **adjacency matrix** A. Considering that the nodes have indices from 1 to n, this matrix of size $n \times n$ is defined as follows: for nodes u and v, A_{uv} is 1 if u and v are adjacent, 0 otherwise. The advantage of this structure is that it allows for constant time adjacency checking. However, neighbourhood listing takes $\Theta(n)$ operations, and edge listing is in $\Theta(n^2)$. The space complexity of this structure is $\Theta(n^2)$, which is problematic for large graphs as they may not fit in the main memory of a standard computer. Mathematically, various graph problems are best described with a matrix representation: the importance of a node can be determined with the eigenvectors (the solutions x of equations of the form $Ax \propto x$), graph embeddings can be obtained from the Laplacian matrix (given by the difference between the diagonal of degrees and the adjacency matrix: $L = Diag(d_1, \ldots, d_n) - A$), etc.

Datasets that represent graphs are often shared as a file containing a list of edges. An **edge list** $e_1, \ldots, e_m$ takes only $\Theta(m)$ space and allows for an edge listing operation in time $\Theta(m)$. However, neighbourhood listing also takes $\Theta(m)$ and adjacency checking takes $\mathcal{O}(m)$. To counteract these inefficient complexities, one technique consists in sorting the edges in a specific order so that the incident edges of a given node are grouped together. Another interest of the edge list structure is that the names of nodes do not have to be

[3] Note that for a maximisation problem, the direction is reversed: a k-approximation with $k \in]0, 1[$ yields a solution y with $kx \leq y \leq x$.

consecutive integers; instead, they can be the name of a person in a social network, or the URL of a webpage in a webgraph.

In order to favour the neighbourhood listing operation, it is recommended to use **adjacency lists**. This structure stores for each node u a list that represents its neighbours N_u. It takes $\Theta(n+m)$ space and allows for edge listing in time $\Theta(n+m)$. Listing the neighbours of u is possible in optimal time $\Theta(d_u)$. Adjacency checking, or testing whether v is a neighbour of u, takes $\mathcal{O}(d_u)$ operations. In practice, the adjacency lists are stored in the *Compressed Sparse Row* (CSR) format, as described in Figure 2.1: all the neighbours are stored contiguously in an array, and each node has a pointer to the beginning of its neighbours in this array. This format is the one that we use throughout the thesis to describe algorithms and to implement them.



(a) Adjacency list (b) Compressed Sparse Row (CSR)

Figure 2.1: **Graph representations.** An adjacency list stores a list of neighbours for each node. In CSR, all the neighbours are stored in a shared array of length $2m$, and each node has a pointer to its first neighbour. As edges are stored in contiguous memory, this is an equivalent but more compact format that allows for faster memory access.

While the graph itself is a non-hierarchical description of the data, all the data structures that are used to represent a graph imply an ordering on the nodes. Indeed, the adjacency matrix requires to choose which node corresponds to the first line of the matrix, the edge list orders the edges in a certain way that creates a hierarchy between them, and adjacency lists imply orderings both in the array of pointers and within each neighbourhood. For this reason, ordering the nodes of a graph is not just a possibility, but a necessity. We will see that real-world networks are distributed with an associated node ordering even though it is not always documented.

Other data structures are often used in algorithms to realise certain tasks. A crucial one is the concept of priority queue: each node is given a priority that may evolve during the execution of the algorithm, and the priority queue has to be able to provide the node of highest priority at any point of the execution. There are three main operations in such a data structure: inserting a node, updating its priority, and finding the highest-priority node. In practice, we use here an implementation based on a binary heap: each node is placed in the tree and can have two branches, which all contain nodes of lower priority. If the tree is balanced, insertion and update take $\mathcal{O}(\log n)$ time, and the priority node is found in $\Theta(1)$ as it is at the root of the tree. Binary heap are efficient in practice, but they imply a log factor that can be avoided with another structure: the bucket queue. When the priority is a bounded integer, for instance a number between 0 and k, each node can be placed in a bucket (a list) that corresponds to its priority, which takes $\Theta(1)$ operations. To update its priority, it suffices to place it in another bucket, in time $\Theta(1)$ as well. A node of highest priority can be found in time $\mathcal{O}(k)$ in the highest non-empty bucket. This structure is particularly important to sort the nodes and obtain, for instance, a degree ordering in $\mathcal{O}(n)$ time (assuming that the degree of each node is known).

2.2 Graph mining on real-world networks

Data mining consists in extracting specific patterns from a real-world dataset in order to gain knowledge about the underlying mechanisms. At the intersection of graph theory and data mining, the field of graph mining addresses problems such as analysing the properties of a network or extracting subgraphs with given properties. Examples of graph mining tasks include detecting communities, identifying central or important nodes, and counting the occurrences of a specific subgraph. Such tasks can help uncover patterns and structures in the networks, providing valuable insights for the fields that generated the data. Graph mining is also interested in simulating real-world networks with synthetic graphs that emulate some of their properties, in order to study their structure in a more controlled way and give insights on problems such as epidemic propagation, cascading failure or information diffusion.

Graph mining tasks are often related to theoretical problems, but they have to be solved in practice on large graph instances. In this context, theoretical complexity is not sufficient to describe the efficiency of an algorithm, as practical execution time is what matters most. Therefore, it is interesting to design algorithms that follow an intuitive design rather than mathematical properties. Such methods are called *heuristics* and are known to perform well on real-world networks. One challenge of graph mining is to explain this success. *Greedy* algorithms, which make the locally optimal choice at each stage, are a type of heuristic often used to tackle problems on real-world networks.

2.2.1 Properties of real-world networks

As mentioned previously, real-world networks have specific properties due to the situation that they represent. The first task of graph mining is to identify such properties. Note that networks of interest tend to have one giant connected component containing most of the nodes, and that isolated nodes with degree zero are generally ignored, enforcing $m \geq n - 1$ for a connected graph.

First of all, real-world networks are usually *sparse*, which intuitively means that each node is only connected to a small portion of the other nodes. In a social network for instance, each individual knows a few hundred individuals among millions of them. Mathematically, sparsity corresponds to $m \ll \binom{n}{2}$, which means that the number of edges is much smaller than the maximum possible number of edges. This is the contrary of a dense graph, where almost all the possible edges exist. Models of sparse graph generally consider that number of edges is within a small multiplicative factor of the number of nodes, noted $m = \Theta(n)$. Sparsity matters for algorithmic complexity: recall that adjacency lists require $\Theta(n+m)$ space, which for sparse graphs results in a linear space $\Theta(n)$, to be compared with $\Theta(n^2)$ for an adjacency matrix. For this reason, the adjacency matrix format is not used in practice for large sparse graphs, even though it can be adapted with sparse matrix representations.

Another common property of real-world networks is their skewed degree distribution: a few nodes called *hubs* have many neighbours, while most of the other nodes have only a few neighbours. In a scientific citation graph for instance, most articles have less than a hundred citations, while the most famous articles can have tens of thousands of them. Researchers like Clauset et al. [2009] have been interested in fitting the skewed degree distribution on mathematical distributions such as a power law where the probability for a vertex to have degree d is $p(d) \propto d^{-\alpha}$.

Real-world networks often have a high level of clustering, meaning that even though most nodes have only a few neighbours, these neighbours are often themselves intercon-

nected. This feature makes the study of triangles particularly interesting for such graphs, which is the topic of Chapter 5. An additional property of real-world networks is similarity, whereby nodes tend to have similar neighbourhoods: if two nodes share some neighbours, they are likely to share many. More generally, these networks tend to have a community structure, which means that nodes can be grouped into communities so that they have many neighbours within the community and few outside of it. In spite of their sparsity, real-world networks have locally dense structures, but they are small: Danisch et al. [2018] show that the densest subgraphs have a few thousand nodes even in graphs with billions of edges, and that the largest cliques only contain a few dozens of nodes.

Watts and Strogatz [1998] and other researchers have observed that real-world networks exhibit a small-diameter property, meaning that the shortest path between any two nodes typically does not require traversing a large proportion of the nodes of the network. The related small-world property describes the average distance between two nodes instead of the maximum distance. These properties are remarkable given that the sparsity and clustering properties might suggest a more local organisation with a larger diameter. The presence of hubs, which connect different regions of the network, can enhance this small-diameter property.

2.2.2 Leveraging network properties

Algorithmic problems can be studied on different classes of graph and their complexity may differ depending on that class. For instance, the minimum vertex cover problem, which is covered in Chapter 6, is NP-complete for general graphs as well as planar graphs, but is in P for bipartite graphs. This is where parametrised complexity comes in, as it examines the complexity of a problem not only in terms of the size of the graph, but also based on certain of its properties.

The specific properties of real-world networks can trigger a reduced parametrised complexity for a variety of graph mining problems. For the problem of listing the triangles of a graph (presented in Chapter 5), Schank and Wagner [2005] point out that the complexity is $\Theta(m^{1.5})$ in the general case, but that for a graph with largest degree d_{max} there exist algorithms with complexity $\mathcal{O}(m \cdot d_{max})$: in graphs with bounded degree, these algorithms have linear complexity $\mathcal{O}(m)$. Similarly, the Bellman-Ford algorithm computes the shortest paths from a node to all the other nodes in time $\mathcal{O}(n^3)$, but a refined expression of the complexity is $\mathcal{O}(\Delta \cdot m)$ where Δ is the diameter of the graph: for the class of graphs with bounded diameter, this algorithm is linear. For the problem of maximal clique enumeration, Eppstein et al. [2013] propose an algorithm that is polynomial if the density of subgraphs is bounded, and show that it is scalable in practice. As a last example, Brach et al. [2016] define a model of graphs with power-law degree distribution for which several common algorithms have a lower complexity than in the general case; in particular, they identify a class of graphs for which the NP-hard problem of finding a maximum clique is in P.

For some other algorithms, parametrised complexity is not yet able to explain the unexpected swiftness of their execution on real-world networks. For example, Danisch et al. [2017] propose a cubic algorithm to find a densest subgraph but they observe that it runs in linear time in practice. The famous community detection algorithm of Blondel et al. [2008] also has cubic worst-case complexity, but it scales to the largest real-world networks. In Chapter 6, we will see that even the exponential algorithm of Hespe et al. [2020] is able to solve the vertex cover problem as fast as linear algorithms on some real-world networks.

Not only can real-world properties impact time complexity and execution time, they can also improve the quality of the results. In the domain of graph compression, Boldi and Vigna [2004] take advantage of the similarity between neighbourhoods to store a description of real-world networks with as little as 2 bits per edge. For the problem of finding a densest subgraph, a linear-time algorithm introduced by Charikar [2000] is known to be a $\frac{1}{2}$-approximation: it is guaranteed to identify a subgraph that is 50% as dense as the densest, but Danisch et al. [2017] point out that it is often above 99% in practice. For the vertex cover problem, Chapter 6 leverages the same type of phenomenon to certify the quality of heuristic results.

2.2.3 Simulating network properties

To understand the interaction between real-world networks and graph algorithms, it can be crucial to generate synthetic graphs that imitate real-world networks. There exist different random models that fix certain properties and let the other ones vary. Letting a parameter of synthetic graphs vary over a given range is a way to test how structural characteristics affect an algorithm, in terms of execution time or result quality. With random models, it is also possible to apply probability theory to compute average-case complexity or expected results on the class of graphs that the model encompasses.

The most famous random graph model is the one of Erdős and Rényi [1960], where only the number of nodes and the (expected) number of edges are fixed. Although this model allows for preserving sparsity of real-world networks, it does not capture other important properties such as the presence of hubs, high clustering, and community structure. In fact, the degree distribution of the model is a bell curve, which forbids hubs, and there is no preference for connecting nodes that share many neighbours, leading to a low clustering.

Another classical model consists in fixing the degree distribution. To obtain a power-law distribution, Barabási and Albert [1999] propose a generative process based on preferential attachment. The more general configuration model is able to maintain any degree distribution by randomising an initial graph while keeping its degree distribution; it has been refined by Newman [2009] to preserve the clustering property and by Karrer and Newman [2010] for the distribution of any other subgraphs. The model of Watts and Strogatz [1998] preserves both a high clustering and the small-world property. In fact, models have been proposed for arbitrary properties through the concept of exponential random graphs (ERGM) [Snijders, 2011]. A review and classification of random models is proposed by Drobyshevskiy and Turdakov [2020].

Apart from the properties that they preserve, random models are supposed to randomise all the other properties. However, this randomisation is not necessarily uniform: among the graphs that have the fixed properties, some are more likely to be generated than others. For instance, the model of preferential attachment only generate trees when it is executed with a specific parameter, even though other non-tree graphs have a similar power-law degree distribution. Tabourier et al. [2011] tackle this issue with a process that repeatedly swaps a set of random edges: the process converges to a uniform distribution over the graphs that preserve certain properties. Stanton and Pinar [2011] address the joint degree distributions of adjacent nodes and Fosdick et al. [2017] study issues of configuration in the configuration model. Van Koevering et al. [2021] propose a model that preserves the core sequence, which is a description of the repartition of density in the graph.

2.2.4 Sources of real-world networks

Networks can be generated by a diversity of real-world situations, but the data is not always easy to gather and organise, especially at the scale of millions of nodes in which we are interested here. In fact, the largest publicly available networks belong to the same few categories. The most common category is that of webgraphs, where a node corresponds to a webpage and a directed edge is a hyperlink from a webpage to another. As the entire web contains tens of billions of pages [4], networks often represent a limited portion of it, for instance the pages of a national domain such as .fr or the pages of a large website like Wikipedia. Another category represents online social platforms: a node corresponds to the account of a person and an edge can be directed to indicate followers or undirected to indicate friendships.

The fact that these two types of networks are the most common is no coincidence. Webgraphs and online platforms both live on the internet, which facilitates their collection and analysis using computers. On the contrary, situations that appear in the physical world can be more costly to turn into an abstract network. For instance, mapping the roads of a continent requires to cross information from different national sources, and discovering the brain network implies biological experiments with microscopic interventions.

Several repositories have been developed by researchers to centralise real-world networks. They also provide information about the source of the data and an analysis of its properties such as the maximal degree, the diameter, the clustering coefficient, etc. In our work, we used the following repositories [5]: SNAP proposed by Leskovec and Krevl [2014], Konect by Kunegis [2013], NetworkRepository by Rossi and Ahmed [2015] and WebGraph by Boldi and Vigna [2004].

2.3 Measuring time in spite of hardware fluctuations

Measuring the execution time of an algorithm is not an exact problem as opposed to the notions of mathematical complexity. Indeed, it is easy to see when an algorithm is orders of magnitude faster than another, but time measurement can be misleading when it comes to a small relative speedup. The issue is that repeating the exact same experiment can lead to different runtimes: executing the same program with the same data on the same machine, we observed variations as high as 50% depending on which other processes run concurrently on other processors. These *fluctuations* impose to take precautions before using time measurements to compare algorithms or implementations. As a patch for this issue, the Python documentation recommends running the algorithm several times and taking only the fastest execution into account, as opposed to the average time: "*the lowest value gives a lower bound for how fast your machine can run the given code snippet; higher values* [...] *are typically* [*caused*] *by other processes interfering with your timing accuracy*" [6].

In order to accurately compare the execution time of different algorithms, it is important to take into account the architecture of the machine. Julien Sopena, a hardware specialist from LIP6 [7], explained that fluctuations can occur at each level of memory management, both within the process and through interactions with concurrent tasks. This

[4] See current statistics on https://www.worldwidewebsize.com.

[5] The repositories can be found at http://snap.stanford.edu/data, http://konect.cc/networks, https://networkrepository.com, and https://law.di.unimi.it/datasets.php.

[6] https://docs.python.org/3/library/timeit.html#timeit.Timer.repeat

[7] https://www.lip6.fr/actualite/personnes-fiche.php?ident=P1369

section summarizes the causes of these fluctuations and how they can be avoided. To minimise the impact of fluctuations, the experiments of this thesis were conducted by booking an entire computational node on the university cluster [8]. This experimental setting helped to reduce total fluctuations to a relative variation of less than 3% over repeated experiments.

The memory of a shared computer is built on several levels that handle the data with different objectives. The external memory (or disk storage) can store terabytes of data and maintain it without electrical power. When a program uses data, it loads it in main memory (or RAM), and it can duplicate parts of it in several levels of cache, which are smaller and faster blocks of memory. Finally, the processor uses registers to handle the variables needed in the computation.

Most often, the computer contains several processing units and runs several processes simultaneously. Because the computer handles a variety of constraints such as load-balancing between processors, fast access to crucial data, and conflicts between threads, time fluctuations can appear at each level of memory and make it hard to measure a precise and consistent execution time for one particular program.

2.3.1 External memory fluctuations

The external memory is the largest type of memory in the machine, but it is also the slowest: retrieving data from external memory implies a latency of several microseconds compared to a main memory access that takes nanoseconds. For this reason, if the running processes do not use all the main memory available, part of what remains is used as a *page cache*: data from external memory is duplicated in the remaining space of the main memory in order to be accessed faster. In other words, the main memory is used as an additional cache level for external memory data. This duplication is transparent to the programs: read and write operations over files that are stored in external memory work as usual, but they take much less time because they operate on the main memory.

Read operations. When running a graph algorithm, the corresponding program needs to load the graph data from a file that is usually stored in external memory. At the time of execution, it is however possible that the data is in fact stored in page cache, in particular if another program has used the same graph data beforehand. Each operation can be microseconds slower if the data is in external memory, due to the latency of such devices. However, the page cache may not contain the appropriate data at each execution, as it can be erased when other processes require more main memory.

To avoid this problem, it is preferable to measure the duration of the algorithm after the page cache has been loaded. For instance, it is possible to run the same algorithm twice: the first execution will serve to load the data from external memory to page cache, and the second will allow for stable time measurement as the data will be read directly from the page cache without as much fluctuations.

Write operations. The page cache is also involved in the writing operations, which are used to output the result of an execution or to log its progression. Even when a file is stored in external memory, a write operation temporarily puts its content in a file of the page cache and only transfers it to the external memory when the page cache is full. If the main memory is large, the page cache can hold several gigabytes of data: the external

[8] MeSU: https://hpcave.upmc.fr

writing becomes a rare and costly action, which causes major fluctuations in time measurements.

To prevent the sudden writing of all the page cache from disturbing the time measurement of a program, it is possible to force the write operations to operate directly in external memory. One option is to disable the page cache, and the other is to write sink instructions, which force the page cache to flush its data into the external memory. In both cases, limiting the fluctuations comes at the cost of a slower global execution.

Concurrent processes with shared external memory. The external memory is connected to the main memory through cables that have a limited data transfer bandwidth. This means that the total debit of read and write operations that can be performed is limited for the whole computer. If another process is transferring large amounts of data from or to the external memory, it limits the ability of other processes to do the same: the latency is shared across processes.

To ensure that other processes do not have a major impact on the read and write operations of the process being timed, it is important to monitor the processes and cancel the measurement if their amount of read and write operations represents a significant portion of the available bandwidth. Some time measurement programs are also able to ignore the latency caused by external memory reads or writes in order to focus on the in-memory computation time.

2.3.2 Main memory fluctuations

To deal with large computational tasks, it can be necessary to use a computer cluster: it is a set of computers called *nodes* that are linked to operate as a single system with large memory and computational power. Physically, the main memory is distributed across the nodes, but programs see it as one consistent block and cannot control where their data is stored. This concept is called a non-uniform memory access (NUMA) and it causes fluctuations: a node can access its data faster if it is stored nearby. Accessing data that is stored in the main memory of other nodes causes a latency and faces another issue of limited bandwidth.

Memory allocation strategies. Several strategies exist to decide where to allocate the memory for a given process on a computer cluster. Some of these algorithms are part of the operating system, while others are built into the hardware, in which case modifying them requires to restart and reconfigure the cluster. In any case, these strategies have to strike a balance between locality and uniformity. The general-use strategy called *interleaving* favours uniformity: it consists in distributing the data evenly across all the physical parts of the main memory; this strategy avoids the worst-case scenario when a process has to retrieve its data from the most remote parts of the memory. On the contrary, the *first-touch* strategy favours locality by allocating the sections of the main memory that are closest to the node that owns the data.

Concurrent processes with shared main memory. Even when our process uses a first-touch allocation strategy, the access time can still be influenced by other processes running on different nodes of the computer cluster. First, these processes may use interleaving, which can occupy memory blocks on our node. As a result, there may be less close memory available for our process, forcing it to store data in more remote parts of the shared memory. Second, the bandwidth between nodes is shared by all the processes, thus the

retrieval of data that is stored further away is impacted by the other processes that run during our time measurements. Booking a full node to ensure that no other process runs on it is not sufficient, as it does not specify where the data is stored. One way around is to force our data to be stored in contiguous memory by activating *huge-pages*: the data is split into pages of a given size (usually 4kB by default) that are scattered in different parts of the memory; setting a larger page size (for instance 1GB) ensures that large amounts of data are stored together and possibly closer to our node.

2.3.3 Cache fluctuations

While the main memory handles data pages of typical size 4kB, the cache handles data lines of typical size 64B. When a process accesses a piece of data stored in a specific address of the main memory, the corresponding data line is duplicated in an address of the cache. The cache address is not chosen at random: it is generally computed from the main memory address by applying a built-in hash function. If the hash function has a certain periodicity, the process may repeatedly paste its data on the same cache address, which hinders the cache efficiency. Other processes can also interfere if a level of cache is shared across several processors.

Cache colouring is the research topic that focuses on reverse-engineering this hash function to ensure that selected main memory areas will match certain reserved areas of the cache. Issues due to concurrent processes can be avoided by booking all the processors that share a level of cache, even if only one is used for computation.

2.3.4 Processor fluctuations

Hyperthreading. To increase the utilisation rate of a pipeline, processors use *hyperthreading*, which consists in executing several threads simultaneously on the same processor. The expected time saving is around 30%. In practice, forcing this extra computation increases the heat production, which requires to reduce the clock frequency to maintain an acceptable temperature. The final time saving is therefore not controllable: one could almost consider that the pace of operations depends on the weather. In particular, if we use a single thread and another process takes another thread of our processor, our execution time will be disturbed. We must therefore book all the associated processors even if we only use one. Doing this may require to know the topology of the cluster and to book contiguous nodes.

False-sharing. Even on a completely isolated machine, instability can be observed between two identical executions. If two variables are stored in the same cache line, changing the value of one will require to reload the cache line for the other. In the case of a parallelised algorithm, this *false-sharing* induces unexpected competition between threads, which depends on the architecture and the current state of the cache. This can be avoided with *padding*, which consists in interposing unnecessary data to ensure that the useful pieces of data do not share the same cache line.

Prefetching. One of the industrial secrets of processors is their *prefetching* algorithm. This is the strategy used to predict the next accesses to data and thus anticipate their retrieval in the nearest memory levels. For example, if an algorithm scans an array, the processor will provide the first few rows of requested data and then anticipate the next few rows to be fetched from memory and stored in a high cache level. This phenomenon

also occurs for more complex operations such as traversing a chained list. It reinforces the importance of grouping correlated data and may explain the difference in results from one machine to another.

Chapter 3

Review of ordering methods

Summary

The idea of ordering a graph goes against the intuitive concept of a graph. Indeed, a graph represents a set of elements and relations that are not subject to any hierarchy: as opposed to a geometric plane where there is a leftmost and a rightmost element, there is no first or last node in a graph. This offers a lot of flexibility to describe real-world interactions that are not bound to a physical space, such as friendships or citations.

However, the file that stores the description of a graph is ordered. The memory of computers is made of sequences of bytes that represent data in a sequential way. To store the graph in memory, a choice has to be made to decide which is the first node and which is the last, which nodes are side-by-side and which are further apart. This unavoidable ordering adds extra information to the graph structure, which has been used in a variety of algorithmic tasks to improve performance, speed or quality.

The current chapter reviews the use of ordering methods in the literature for graph algorithms. We identify the main applications that triggered discoveries of new graph orderings, and give a classification of the different underlying mechanisms and algorithmic techniques. For practical purposes, we also discuss the trade-off between the time needed to compute an ordering and its impact on the algorithm.

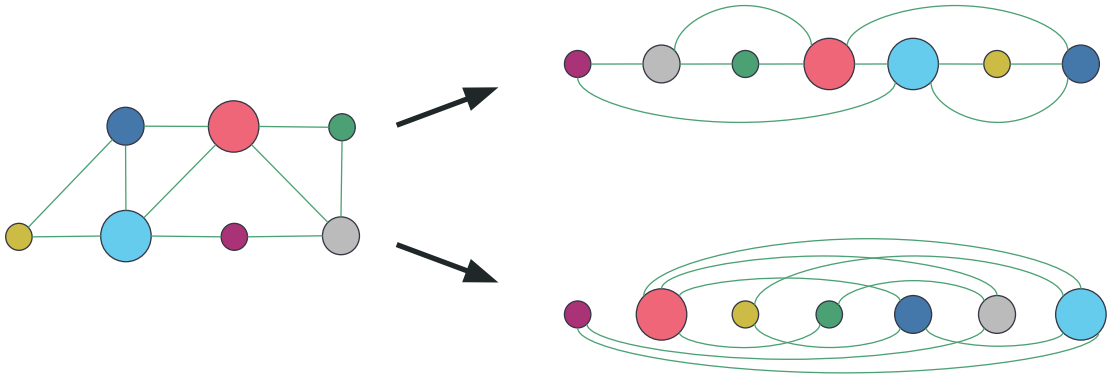


Figure 3.1: **Examples of different node orderings for the same graph.** Node sizes represent their degree and help distinguishing them. The initial graph has no particular ordering, the top ordering has been designed to minimise the length of edges, and the bottom one maximises them.

3.1 Introduction

A node ordering, also called vertex ordering, consists in defining a sequence in which the nodes are organised from first to last, as opposed to the graph structure itself that does not contain such information. Figure 3.1 shows a toy graph with seven nodes and two possible orderings: the top one aims at bringing closer in the ordering the nodes that have an edge in common; the bottom ordering does the opposite, which makes the edges look longer. Here is the formal definition that we choose for a node ordering:

Definition 1 (Node ordering) *Given a graph $G = (V, E)$, a node ordering is a bijection $\pi : V \to [\![1, n]\!]$. When $V = [\![1, n]\!]$, π is a permutation over V. The index of a node $u \in V$ in this permutation is noted π_u. It defines a total order $\prec$ over the nodes, given for nodes $u, v \in V$ by $u \prec v \iff \pi_u < \pi_v$.*

When the graph is initially undirected, which is the case in this thesis if not stated otherwise, then an ordering π creates an artificial orientation of the edges following the values of π: an undirected edge $\{u, v\}$ becomes a directed edge (u, v) if $\pi_u < \pi_v$, and (v, u) otherwise. Now the neighbours N_u of u are partitioned into predecessors N_u^- and successors N_u^+, and u has an in-degree $d_u^- = |N_u^-|$ and an out-degree $d_u^+ = |N_u^+|$.

This chapter is a review of influential node orderings that have been designed to improve the algorithms for a variety of tasks. In Section 3.2, we propose a classification of orderings according to the underlying mechanism (Section 3.2.1) and the algorithmic method (Section 3.2.2). Section 3.3 presents the node orderings by analysing different domains of application and showing the impact that orderings can have, and Section 3.4 points out standard graph problems that can be reformulated as an ordering problem. We develop the issues related to the computation time of orderings in Section 3.5.

3.2 Classification of node orderings

In spite of their diverse application domains, node orderings share some principles that this section aims to classify. On the one hand, orderings are inspired by a few underlying mechanisms, such as bringing nodes close in the ordering when they are close in the graph, or placing important nodes at a specific position of the ordering. On the other hands, the process of computing an ordering uses general algorithmic methods such as optimisation or streaming designs. Table 3.1 presents the classification of all the orderings that will be described in the following sections.

3.2.1 Underlying mechanisms of node orderings

Node orderings can be designed to enhance certain properties of consecutive nodes or groups of nodes. While the applications and algorithms to obtain orderings are diverse, we propose to classify ordering methods based on their underlying mechanism. These definitions will be used to tag ordering methods, as summed up in Table 3.1.

The mechanism of **locality** aims at linking proximity in the ordering to proximity in the graph. Two opposite implications may be desired in the design of orderings, and they correspond to slightly different mechanisms: we call *graph locality* the mechanism whereby neighbouring nodes have close indices in the ordering, and *index locality* the mechanism where nodes with consecutive or close indices are likely to be neighbours in the graph. Related to locality is the mechanism of **similarity**, which ensures that nodes

Ordering name	Mechanism					Algorithmic method				Section
	Locality	Similarity	Centrality	Grouping	Precedence	Objective function	Priority selection	Graph reduction	Streaming	
Natural	?	?	?	?	?					3.3
Random										3.3
BFS	✓						✓			3.3
DFS	✓						✓			3.3
RCM	✓		✓				✓			3.3.4
Degree			✓							3.3.1
Closeness			✓							3.3.1
Betweenness			✓							3.3.1
Pagerank			✓			✓				3.3.1
Core			✓		✓		✓			3.3.2
Peeling			✓				✓			3.3.2
Density friendly			✓			✓	✓			3.3.2
Colouring				✓						3.3.3
Largest-first			✓				✓			3.3.3
Split-degree			✓						✓	3.3.3
Gscore	✓	✓				✓				3.3.4
Gorder	✓	✓					✓			3.3.4
Rabbit				✓		✓				3.3.4
Lightweight			✓					✓		3.3.4
Bandwidth	✓					✓				3.3.4
MinLinA	✓					✓				3.3.4
MinLogA	✓					✓				3.3.5
MinLogGapA	✓					✓				3.3.5
BiMinLogGapA	✓					✓				3.3.5
Shingle		✓						✓		3.3.5
Recursive bisection	✓			✓				✓		3.3.5
Slashburn			✓				✓			3.3.5
LLP				✓						3.3.5
Metis				✓				✓		3.3.6
LDG	✓			✓					✓	3.3.6
Fennel	✓			✓					✓	3.3.6
Vebo				✓					✓	3.3.6
Sheep				✓				✓		3.3.6
Personalised Pagerank			✓			✓				3.3.6
Balanced					✓	✓				3.3.7
Revorder					✓	✓				3.3.7
C^{++} and C^{+-}					✓	✓				Ch. 5
Neigh					✓					Ch. 5
Check			✓		✓				✓	Ch. 5

Table 3.1: **Classification of node ordering methods.** Mechanisms represent the properties that the ordering pursues, and they are described in Section 3.2.1. Algorithmic methods are the way in which the ordering is obtained, as explained in Section 3.2.2.

with close indices have similar neighbourhoods, even though they may not be neighbours themselves.

It is also possible to order the nodes according to a **centrality** measure, which is a certain property that describes the importance of each node in the network. The mechanism of centrality consists in placing the nodes of high centrality in a specific position, for instance at the beginning of the ordering. A usual technique is to rank all the nodes according to their centrality, but other designs are possible such as ranking only the nodes of highest centrality or placing them at either extremity of the ordering. To construct an ordering π from a centrality measure $c : V \to \mathbb{R}$, two things have to be taken into account. First, c takes values in $\mathbb{R}$ instead of $[\![1, n]\!]$ for orderings: the range and concentration of the values of c is lost since the ordering only retains the rank of each node with respect to the others. Second, several nodes may have the same centrality, which requires a tie-breaking strategy. Usual strategies are to break ties at random, or following the natural ordering given by initial node indices: $\pi_u < \pi_v \iff \big(c(u) < c(v)$ or $(c(u) = c(v)$ and $u < v)\big)$. The tie-breaking strategy can be crucial in the case where many nodes have the same score.

The **grouping** mechanism consists in constructing an ordering from groups of nodes with a given property. The groups can be defined as highly connected subsets of nodes, in which case the literature on community detection can be leveraged, but other constraints can be defined: imposing a size on the groups, taking group of independent nodes, etc.

Lastly, there exist orderings that constrain the number of predecessors and successors of nodes in the ordering, which we call a **precedence** mechanism. An example of such ordering is one that ensures that the number of successors of each node is bounded by a constant.

3.2.2 Algorithmic methods to obtain node orderings

There are several ways to compute a node ordering, and researchers have developed various algorithms to achieve this task. We can broadly classify these methods into four categories based on their approach: objective function optimisation, priority selection, graph reduction and node streaming. The choice of method depends on the specific problem at hand and it is not necessarily determined by the underlying mechanism or by the application domain.

In the literature, node orderings are sometimes described as an **objective function** that formalises the desired properties of the ordering. These functions are sometimes referred to as layout problems, as surveyed by Díaz et al. [2002]. Optimising these objective functions leads to orderings that satisfy the properties, but it is often NP-hard to find such an ordering. It is thus useful to design heuristics that improve the value of the objective function even though they do not reach the optimum. For this reason, we distinguish between the ordering defined as an optimum of the objective function and an ordering obtained from a heuristic that addresses the optimisation. There exist approaches called meta-heuristics that can in principle be used for any of these functions. For instance, Wei et al. [2016] use *simulated annealing*, a meta-heuristic that starts from an ordering and swaps the indices of random nodes with some probability. The probability is higher if the swap improves the objective function, but it is still non-zero otherwise: intuitively, a bad choice can be made in the hope that it later leads to a better solution.

Alternatively, an efficient way to obtain scalable approximations is to design a greedy algorithm: such a procedure will gradually improve its solution by taking the most beneficial step. For NP-hard optimisations, there is no guarantee that the result is optimum, but the time complexity can be linear or quasi-linear. Greedy methods can be easier to design, to program, and to tune; as shown in Chapter 6, they can give fast and competitive

results on real-world data, which is why they are often used in practice.

The algorithmic method of **priority selection** constructs the ordering by defining a priority for each node, then selecting one of highest priority, and updating the priority of the nodes that are still to be selected. This situation equates the ordering with a priority function over the nodes $\nu : V \to \mathbb{R}$ that evolves during the execution of the algorithm. The first examples of this type of ordering are graph traversals, where only one node is visible at the beginning, and visiting it triggers an update of the visible graph and impacts the subsequent order of nodes. Priority selection also arises as a variation of centrality measures when the ordering is built by selecting a node of highest centrality, removing it from the graph, and recomputing the centrality of remaining nodes. Such procedure is also called peeling.

To process a graph and obtain an ordering, several papers use a **graph reduction**, which consists in removing or merging nodes and edges to obtain a smaller graph. The criteria for the reduction are diverse: keeping only the edges of a spanning tree, ignoring the edges that cut a node partition, considering only the nodes of high centrality... The ordering is then computed on the smaller graph, and later extended to include all the original nodes.

Lastly, some node orderings are obtained with **streaming** algorithms: nodes are examined as a sequence and they have to be placed in the ordering with limited information on the rest of the graph. Usually, such algorithms allow for a sub-linear amount of memory, for instance $\mathcal{O}(1)$ or $\mathcal{O}(\log n)$, and they specify the number of passes that are allowed. In particular, one-pass algorithms can only read the sequence of nodes once before giving their output. For graph algorithms, there may be extra information attached to the nodes while reading it, such as their degree or their list of neighbours.

3.3 Node orderings and their application domains

↑ back to Table 3.1 ↑

Node orderings extend the graph data-structure and operate a link between the mathematical description and the hardware properties. For this reason, they are used in a wide range of contexts. In this section, we walk through some of these application domains and present node orderings that have been designed to tackle the corresponding problems. The orderings are named in **bold font** for easier reference.

Let us start with the most notable types of ordering, which are used across all the domains of application: the natural ordering obtained during the data collection, a random ordering, and a graph search ordering.

Natural or original ordering. The natural ordering of a graph is the one that is provided in the initial file: standard storage formats for graphs (adjacency list or matrix, incidence matrix, edge list) need to give each node an identifier, which is usually an integer between 1 and n. These numbers are arbitrary in the sense that the dataset providers seldom reveal which order they followed to scrap the data. Yet, it is not a random ordering, as their may be an unknown underlying mechanism. Let us consider the use case where one wants to create a dataset that represents a webgraph. Starting from a few seed webpages, a crawler follows hyperlinks to discover more and more pages. In which order will the pages be stored in the resulting graph? If they are stored according to their time of discovery, then the ordering of the graph follows a traversal pattern as described further; if they are stored in alphabetical order of their URL, nodes of consecutive indices are likely to represent pages of the same website, and thus more likely to share an edge. The latter

situation enables Boldi and Vigna [2004] to compress webgraphs without explicitly using orderings.

Random orderings. A random ordering aims at erasing the information that may be stored in the natural ordering of the network. Using randomised orderings as benchmark is key to study order-dependant algorithms, because they reveal the behaviour of the algorithm when no advantage is given by the ordering. Note also that drawing several random orderings allows for measurements of the variance of the algorithm and other statistical measures.

BFS and DFS orderings. Graph traversal algorithms consists in visiting all the nodes of a graph in a given order [Cormen et al., 2009]. *Breadth-first search* (BFS) starts from a root node, then visits all its neighbours, then all the neighbours of its neighbours, etc; it creates successive levels in which all the nodes are at the same distance of the root node. *Depth-first search* (DFS) starts from a root node and follows one path as far as possible before backtracking. In both cases, the underlying mechanism is locality as nodes that are close in the ordering are also close in the graph. The algorithmic method is a priority selection, and it is possible to obtain multiple different orderings depending on the selection of the root node and the priority among neighbours. These orderings come from standard graph algorithms and can be used as benchmark to evaluate other methods. They are also likely to correspond to the natural ordering when networks are discovered by following edges from known nodes.

3.3.1 Network robustness against attacks

↑ back to Table 3.1 ↑

The idea of network robustness is to test the connectivity of the remaining network after a number of nodes or edges have been removed. The removal process is called a *failure* if the elements are removed at random, and an *attack* if the elements are removed following a specific ordering, usually based on a centrality measure. In the case of an attack, the ordering is always crucial, but the attack strategy also depends on the metrics that is used to measure the connectivity of the remaining network: usual metrics are the size of the largest connected component, the diameter of the graph, or the average length of shortest paths. We are here interested in the orderings involved in attacks that remove nodes; note that any type of centrality ranking can be used, several of which are studied by Vigna and Boldi [2014].

Degree centrality. Sorting the nodes according to their degree is possibly the most widespread type of node centrality. In robustness studies, it is quite natural to study the resilience of a network where nodes of highest degree are removed. Albert et al. [2000] find that scale-free networks are highly robust to random failures but vulnerable when it comes to attacks on the nodes of high degree. Magnien et al. [2011] mitigate this result highlighting the fact that removing nodes of higher degree boils down to removing many edges; measuring the effort of attack in terms of edges instead of nodes makes the difference between random failure and degree attack less significant.

Closeness centrality. The closeness of a node is the inverse of the average distance that separates it from other nodes [Bavelas, 1950]. More formally, the closeness of a node u is $c(u) = \frac{n-1}{\sum_{v \in V} d(u,v)}$, where $d(u, v)$ is the minimum length of a path between u and v. Iyer et al. [2013] test the robustness of synthetic and real-world networks against various

types of centrality attacks. They find that degree attacks are the more damaging in the context where the ordering is pre-defined and cannot be updated. On the other hand, in a sequential attack where centralities are updated, all the attacks have the same efficiency in random graph models, but the most efficient for real-world networks are the closeness and betweenness attacks.

Betweenness centrality. The betweenness of a node is the proportion of shortest paths (between two other nodes) that go through this node [Freeman, 1977]. Formally, the betweenness of v is $b(v) = \sum_{u \neq v \neq w} \frac{\sigma_{uw}(v)}{\sigma_{uw}}$, where σ_{uw} is the number of shortest paths from u to w, and $\sigma_{uw}(v)$ is the number of such paths that go through v. Holme et al. [2002] analyse different types of networks and observe that a betweenness attack can be more efficient than a degree attack, especially on real-world networks where the two centralities do not show the correlation that they have for some random graph models. Another factor is the update of the centrality: an attack is more efficient if, after each removal, the centrality is re-computed in the remaining network. As this update can be costly in terms of computation, doing it is also an experimental choice.

Eigenvector centrality and Pagerank. Eigenvector centrality has also been used as a measure for robustness [Ellens and Kooij, 2013]. The centrality of all the nodes is given by a vector x that satisfies $Ax = \lambda x$, where λ is a constant and A is the adjacency matrix of the graph. One famous variation of eigenvector centrality is the Pagerank introduced by Page et al. [1999]. It represents the rate of visit of each node during a random walk with a damping factor α, given by a vector x that results from the equation $x = \alpha x W + (1 - \alpha)y$, where W is the probability transition of the random walk and y is the constant vector $(\frac{1}{n} \dots \frac{1}{n})$. Bressan et al. [2018] propose a method to estimate the Pagerank of a node while visiting only a fraction of the graph. In case of an evolving network, Pagerank has been leveraged by Huynh et al. [2014] to create *Liverank*, a centrality measure that describes how likely a node is to still exist in the graph.

3.3.2 Identification of dense subgraphs

↑ back to Table 3.1 ↑

Real-world networks are generally sparse but tend to have areas of high density due to their skewed degree distribution and their clustering properties. Finding these dense areas is a key algorithmic problem. Several definitions of density compete, and a variety of orderings have been proposed as heuristics to find subgraphs of high density.

Given a graph $G = (V, E)$, the general density problem consists in finding a subset of nodes $W \subseteq V$ that maximises the density $\rho(W)$ of the induced subgraph. Density functions ρ are based on $E_W = E \cap W \times W$, the set of edges within W. A first definition consists in computing the ratio between the number of edges in E_W and the maximum possible number of edges among $|W|$ nodes: $\rho_1(W) = |E_W| / \binom{|W|}{2}$. However, this metric can be counter-intuitive as a set of two nodes with one edge achieves maximum density $\rho_1 = 1$. Another definition computes the ratio between the number of edges and the number of nodes: $\rho_2(W) = |E_W|/|W|$. Goldberg [1984] provides a polynomial algorithm that minimises ρ_2 using max-flow algorithms, but numerous heuristics have been designed for faster results, as surveyed by Lee et al. [2010] and Fang et al. [2022]. We are here interested in the ones that involve node orderings.

Core ordering. The key idea of the core ordering is to remove nodes of low degree until the remaining *core* is dense. This priority selection is also referred to as *smallest-first* or

degeneracy ordering. Nodes are first ranked by their total degree, then the lowest-ranking nodes are removed and the degrees and ranks are updated. The k-core is defined as the first subgraph of the decomposition where all the nodes have degree at least k. The degeneracy of the graph is the highest value γ such that the γ-core is not empty; this subgraph is called the core of the graph. The core ordering can be seen as a precedence mechanism as it ensures that nodes have at most γ successors in the ordering. Asahiro et al. [2000] first introduce this peeling heuristic to address the density problem with fixed size $|W| = x$ in weighted graphs: they remove nodes whose edges have low weight until exactly x nodes remain. The procedure is then adapted to the ρ_2-density problem by Charikar [2000], who measures the density of the remaining subgraph at each step, and outputs the one that had the highest density. This linear heuristic is proven to be a 2-approximation of the densest subgraph problem, which means that the densest subgraph is at most twice as dense as the resulting subgraph. Charikar also adapts the method to directed graphs with a more general definition of density.

Generalised peeling orderings. The peeling procedure described for the core ordering can be extended in two directions. First, instead of selecting the node of lowest degree at each iteration, it is possible to use another centrality. In their review, Malliaros et al. [2019] mention several centrality measures such as the number of neighbours at a given distance, the number of cycles of fixed length, etc. In fact, Batagelj and Zaveršnik [2002] prove that any function can be used as long as it is monotone, which means that its value can only decrease for remaining nodes when one node is removed. Second, other patterns may be taken into account to create the ordering. The main example is the k-clique densest subgraph problem by Tsourakakis [2015], which consists in finding a subset with as many k-cliques per node as possible. Interestingly, an efficient algorithm for this problem by Danisch et al. [2018] uses the core ordering to accelerate its execution.

Density friendly ordering. After observing that the core of the graph is not necessarily the densest subgraph of the core decomposition, Tatti and Gionis [2015] propose a different procedure to correct this flaw. It starts by finding the exact densest subgraph using a max-flow algorithm; then, it removes this subgraph and the edges that connected it to the rest of the graph are transformed into self-loops; these steps are repeated until the graph is empty. This algorithm creates an ordering that follows a density decrease: the subgraph induced by a short prefix is denser than the subgraph induced by a longer prefix. The mechanism is a priority selection, but Danisch et al. [2017] reformulate it as an objective function in the form of a quadratic program. They provide a scalable algorithm that computes the same ordering using convex optimisation, and they call it the density friendly ordering.

3.3.3 Pattern mining

↑ back to Table 3.1 ↑

Patterns designate a subgraph of small size with a precise combination of edges; they are also called motifs or graphlets. Problems of pattern mining can involve one specific pattern, or all the patterns of a given size. The task differs whether the pattern has to be an induced subgraph or not: an induced subgraph consists of a set of nodes and *all* the edges that they share, while a general subgraph may ignore some edges. Beyond the diversity of patterns, mining also encompasses several tasks: a counting algorithm simply outputs the number of patterns found in the graph (possibly with an accepted margin of error), a listing algorithm enumerates all the patterns with the nodes that are

involved, and sampling algorithms yield a (possibly random) selection of patterns. Many of these problems comes with a large body of literature. We study one of them in depth in Chapter 5: the problem of enumerating induced triangles.

Node orderings are a key technique in pattern mining: processing the nodes in a specific order can help finding the patterns with less operations. The most widely used orderings are those that bound the out-degree of the nodes: the degree ordering [Chiba and Nishizeki, 1985, Latapy, 2008, Ortmann and Brandes, 2013, Pinar et al., 2017, Arifuzzaman et al., 2019] and the core ordering [Schank and Wagner, 2005, Danisch et al., 2018, Turk and Turkoglu, 2019, Pashanasangi and Seshadhri, 2021, Bressan and Roth, 2021]. They appear in works on different types of graphs and on different patterns, as well as for other mining tasks such as the enumeration of dominating sets [Kurita et al., 2018]. Other orderings have been used for specific research questions:

Parallel core ordering. The core ordering requires to update the degrees of the remaining nodes after each deletion of a node. This is problematic in a parallel execution, as the processors will need much communication to share these updates. Shi et al. [2021] propose a variation of the core peeling method: instead of removing the node of lowest degree at each iteration, they remove a constant fraction of the nodes with lowest degree, following an algorithm by Goodrich and Pszona [2011]. The resulting ordering method is *work*-efficient, which means that its total asymptotic number of operations matches the sequential algorithm, and has a logarithmic *span*, which means that its execution time is logarithmic if there are enough parallel processors. Hu et al. [2021] use a similar ordering for pattern counting on GPU.

Largest-first ordering. Core ordering is also called smallest-first because the peeling repeatedly removes the node of smallest degree. The *largest-first* ordering is the priority selection that repeatedly removes the node of largest degree; also called *smallest-last* by Matula and Beck [1983], it is not equivalent to reversing the core ordering. Bressan [2021] propose an algorithm that samples patterns uniformly using this ordering to compute the likelihood of a given pattern.

Colouring ordering. In their work on clique listing, Li et al. [2020] apply a pre-routine where each node is coloured so that neighbours have distinct colours. Noticing that all of the nodes of a clique have distinct colours, the authors are able to prune the search space when too few colours are available; this accelerates the execution of the listing algorithm. The colouring ordering follows a grouping mechanism where groups are independent sets. It is obtained with a greedy colouring algorithm that is itself based on a degree or core ordering as heuristics to obtain a small number of colours.

Split-degree ordering. In their study of the asymptotic cost of listing triangles, Xiao et al. [2017] note that the choice of ordering depends on the choice of algorithm. Using a model of random networks, they find that it is optimal to use a Round-Robin, or *split-degree* ordering, that ensures that nodes of large degree are either at the beginning or at the end of the ordering. This streaming algorithm based on degree centrality is one ordering that we use in Chapter 5.

3.3.4 Cache optimisation

↑ back to Table 3.1 ↑

Cache optimisation consists in adapting the data or the algorithms to benefit from the cache architecture. Broadly speaking, a computer is made of memory units that store the data and a processor that executes the instructions of an algorithm on this data. The location of data is subject to a trade-off between access speed and storage space: the execution is faster if the data is close to the processor, but there is not enough physical space near the processor to store all the data. A cache system is a set of smaller memory units that are located closer to the processor; contiguous blocks of data are duplicated in cache for faster access. More details and a description of a cache system with several levels are given in Chapter 4.

A graph algorithm benefits from cache systems when it processes several pieces of data that are stored contiguously. In spite of their huge diversity, many graph algorithms rely on the same basic operation: iterating over the neighbours of a node. For a graph that is stored as adjacency lists, the cache system is most efficient when lists that are required successively are stored contiguously. As an example, consider a subpart of a depth-first search algorithm: starting from a node u, it reads a neighbour $v \in N_u$, and proceeds with a neighbour $w \in N_v$. During the execution, the processor will first fetch the block containing N_u from the main memory and store it into the cache; if N_u and N_v are stored contiguously, it is likely that N_v is already in the cache, which makes the access to w faster. Otherwise, there is a *cache-miss* and N_v also has to be retrieved in main memory, which causes a delay. Ailamaki et al. [1999] measure that this delay takes more than half of the total execution time of some database algorithms.

Node orderings with locality or similarity mechanisms can help to ensure that nodes that are close in the graph are also close in memory. Several objective functions have been proposed to represent these properties and Barik et al. [2020] propose an experimental comparison of some of them. While the details of the cache architecture depend on the manufacturer and model of the processor, the objective functions are mathematical guidelines to improve the locality, and with it the cache performance of any processor.

Bandwidth. The graph bandwidth problem consists in minimising the objective function $\max_{\{u,v\}\in E} |\pi_u - \pi_v|$; it is called bandwidth because it corresponds to the width of non-zero values along the diagonal of the adjacency matrix. A small bandwidth means that neighbour nodes are never too far in the ordering π, which corresponds to a mechanism of graph locality and may prevent cache-misses. Finding an ordering with minimum bandwidth is NP-hard [Garey et al., 1976].

RCM ordering. The reversed Cuthill and McKee [1969] ordering (RCM) is a linear-time heuristic that addresses the bandwidth minimisation problem. It is a refined version of breadth-first search where nodes of a given search level are ranked by their degree, and thus belongs to the category of graph traversal algorithms.

MinLinA. The minimum linear arrangement problem consists in finding a node ordering π such that the objective function $\sum_{\{u,v\}\in E} |\pi_u - \pi_v|$ is minimised. This is another way to formalise the mechanism of locality: π ensures that neighbours have close indices on average. Initially, Harper [1964] introduces this problem on hypercube graphs for error-correction, and Garey et al. [1976] later show that it is NP-hard.

Gscore and Gorder. To build an ordering that applies to more general graph algorithms, Wei et al. [2016] consider a directed graph and propose to optimise a cost that takes "sibling" relationships as well as neighbourhood into account. Pairs of nodes (u, v) are given a proximity score defined as $S(u, v) = S_s(u, v) + S_n(u, v)$, where $S_s(u, v)$ is the number of times nodes u and v coexist in sibling relationships (number of common predecessors) and $S_n(u, v)$ is the number of times they are in a neighbour relationship (either 0, 1 or 2 since both directed edges $u \to v$ and $v \to u$ may exist). These two terms stand for locality and similarity. The objective is defined as finding an ordering π that maximises the objective function Gscore, defined by $\sum_{0<\pi_u-\pi_v\leq w} S(u, v)$, where w is a fixed parameter (window size): two nodes that are at distance less than w in π should have a high proximity score. After proving that the maximisation is NP-hard, the authors propose a greedy $\frac{1}{2w}$-approximation to obtain an ordering that they call Gorder. It is obtained with a priority selection heuristic, where each step selects the node that has the highest proximity with the nodes of the window. The authors report a significant reduction in the cache-miss rate; we reproduce their experiments in Chapter 4.

Rabbit ordering. Following the mechanism of grouping, Arai et al. [2016] base their algorithm on community detection. Building upon the work of Blondel et al. [2008], they use a modularity-based optimisation to obtain a hierarchical community structure. Transforming this hierarchy into an ordering allows them to enhance each level of cache: the higher cache level can store a few nodes that belong to a dense community and thus have a high proximity, while lower levels store more nodes that belong to a sparser community. This ordering claims a similar speedup to Gorder in the execution of various graph algorithms, but with the major advantage of a faster and more scalable ordering procedure. To keep the community structure while optimising the cache even further, Nguélé et al. [2017] present a method that consists in applying Gorder to reorder nodes within the communities.

Lightweight reordering. The performance gain that cache-friendly orderings cause needs to be balanced with the time spent to compute this ordering, called *overhead* (more about that in Section 3.5). Balaji and Lucia [2018] point out that the long ordering phase of Gorder is only balanced if the algorithm is executed thousands of times. The overhead of Rabbit is less deterrent, but it is still not negligible. Noting that a simple degree ordering is efficient to reduce cache-misses, Balaji and Lucia design the HubSorting and HubClustering orderings that consist in gathering the nodes of high degree (the hubs) at the beginning of the ordering. Indeed, these nodes are often required in the execution of an algorithm, and the cache can be more efficient if it already contains their associated data. These two orderings are based on a centrality measure and they involve a graph reduction where only nodes of high centrality are considered.

3.3.5 Graph compression

↑ back to Table 3.1 ↑

Node orderings have been identified as a powerful tool to reduce the storage space of compressed structures that represent graph. The largest publicly available real-world graphs have billions of nodes and tens of billions of edges, which makes their storage and manipulation complicated on most computers; in particular, they can exceed the size of the main memory. As an alternative to off-memory algorithms, which consist in loading only one part of the graph at a time, it is possible to work with compressed structures that

allow for specific graph operations without decompressing the full dataset. This technique relies on the observation that algorithms do not need the graph itself, rather the ability to do some specific operations on this graph. Thus, after identifying these operations, the input graph can in principle be compressed into a lightweight structure. A thorough review on lossless graph compression techniques is presented by Besta and Hoefler [2019].

A famous compression framework has been proposed by Boldi and Vigna [2004]: adjacency lists are compressed by several techniques that leverage the properties of real-world networks, in particular webgraphs where nodes are webpages ordered according to the lexicographic order of URLs. They observe a locality property, which allows them to use a delta-encoding for adjacency lists: if node 50 has neighbours $(51, 52, 53, 54, 55, 65)$, this encoding stores only stores the gaps $(5, 9)$, meaning that the neighbours are the next 5 indices and the next one comes after a gap of 9 nodes. The similarity property of the webgraphs allows the authors to use references between adjacency lists: if two consecutive nodes have the same neighbours, then only the first node stores them in its adjacency list, and the second node only stores a pointer to that list. For webgraphs, the efficiency of the compression is partly due to the fact that the natural ordering of datasets preserves these properties.

On other types of real-world networks, in particular social networks, locality and similarity are not as strong in the natural orderings, which hampers the compression scheme. Apostolico and Drovandi [2009] suggest to apply a BFS ordering as a pre-routine in order to obtain some locality. Chierichetti et al. [2009] show that a favourable ordering can be constructed by optimising the following objective functions, MinLogA and MinLogGapA, two variations of MinLinA with a locality mechanism.

MinLogA. The minimum logarithmic arrangement problem consists in finding a node ordering π such that the objective function $\sum_{\{u,v\}\in E} \log |\pi_u - \pi_v|$ is minimised. The interest of this formula is that the log stands for the number of bits needed to store an edge in the compression framework; an ordering that minimises it is likely to lead to a better compression rate. Safro and Temkin [2011] propose a generalisation of this problem to edges with weight and nodes with volume. Chierichetti et al. [2009] show that MinLogA is NP-hard and that orderings that minimise MinLinA do not necessarily minimise MinLogA.

MinLogGapA. To further align the minimisation problem with the details of the delta-encoding, one can find an ordering π that minimises the objective function $\sum_{u\in V} \sum_{k=1}^{d_u-1} \log |\pi_{u_{k+1}} - \pi_{u_k}|$, where $u_1, \ldots, u_{d_u-1}$ are the neighbours of u sorted according to π. Though this minimisation problem reflect the characteristics of the compression framework and could lead to high compression rates, finding an optimum ordering is not realistic on large graphs as Dhulipala et al. [2016] proved that MinLogGapA is also NP-hard.

Shingle ordering. Instead of pursuing the minimisation of MinLogA or MinLogGapA, Chierichetti et al. [2009] propose a scalable heuristic based on similarity. For a random permutation σ over the nodes, the *shingle* of a node u is $s(u) = \min_{v\in N_u} \sigma_v$, or the smallest neighbour of u according to σ. The shingle ordering consists in sorting nodes according to their shingle $s(u)$; ties are broken with the second shingle, defined as the second smallest neighbour according to σ. The probability that two nodes u and v have the same shingle

is given by the Jaccard index $J(u,v) = \frac{|N_u \cap N_v|}{|N_u \cup N_v|}$: if u and v share a large portion of their neighbourhood, they are likely to have the same shingle and hence to be close in the ordering. This algorithm can be seen as a graph reduction where only edges between a node and its shingle are considered.

BiMinLogGapA. A generalisation of the reordering techniques for graph compression is proposed by Dhulipala et al. [2016]. They introduce the BiMinLogGapA objective function, that is similar to MinLogGapA except that it relies on a bipartite graph: using the vocabulary of databases and inverted indexes, they distinguish the set of *query* nodes Q and *data* nodes D. This is a usual way of representing an inverted index (a relation between a query word and all the documents that contain it) used in particular by web browsers. The minimisation problem consists in finding an ordering π over D that minimises $\sum_{q \in Q} \sum_{k=1}^{d_u - 1} \log |\pi_{q_{k+1}} - \pi_{q_k}|$, where $q_1, \ldots, q_{d_q}$ are the neighbours of q sorted according to π, which all belong to D as the graph is bipartite. While using bipartite graphs seems like a special case, it is in fact a generalisation as any instance of either MinLogA or MinLogGapA can be translated into an instance of MinLogGapA. This minimisation is therefore NP-hard as well.

Recursive bisection. Dhulipala et al. [2016], who target networks with billions of nodes, propose the following heuristic to reduce the BiMinLogGapA cost. The bisection is a graph reduction that consists in partitioning the data nodes of D into two sets D_1 and D_2 of equal size while reducing the cost. The algorithm creates the partition with the following steps. First, it initialises the partition at random or according to the shingle ordering. Second, it computes for each node $u \in D_1$ the gain $g(u)$ for BiMinLogGapA if u were in D_2 instead; conversely for nodes in D_2. Then, as long as there exist $u \in D_1$ and $v \in D_2$ that have positive gains, it swaps them and thus decreases the BiMinLogGapA cost. Finally, once all swaps have been performed, it pursues to the next step of recursion: sets D_1 and D_2 are also partitioned into two. The recursion stops with sets of one node, which amounts for $\log n$ recursive levels and an overall log-linear time complexity. While this method only orders the data nodes to enhance the locality property, Danisch et al. [2022] propose to adapt it to order the query nodes as well, thus favouring the similarity property in bipartite graphs.

Slashburn ordering. If the graph is stored as an adjacency matrix of size $n \times n$, that consists of blocks of size $b \times b$ (with $b < n$ that divides n), the compression problem can be reformulated as finding a permutation of the row and column indices so that few blocks have non-zero values. Bandwidth minimisation or recursive bisection orderings tend to fill the blocks around the diagonal, but Lim et al. [2014] claim that exploiting this type of community structure is not sufficient in real-world networks that lack small cuts. Instead, they propose to leverage their skewed degree distribution to order hubs and spokes (nodes of high and low connectivity) separately. Slashburn is a priority selection where each iteration goes as follows: remove the k nodes of highest degree (hubs), place them at the beginning of the ordering, and place at the end all the nodes that have been disconnected from the main connected component (spokes). Using cost functions that reflect the matrix encoding, the authors show that Slashburn ordering compresses better than other orderings, although it may not be the case with another encoding such as the one of Boldi and Vigna [2004].

LLP ordering. To improve the compression rate of the webgraph framework when the natural ordering has limited locality and similarity, Boldi et al. [2011] propose to use a hierarchical community detection method and then apply the compression over the resulting ordering of the nodes. They choose the label propagation algorithm of Raghavan et al. [2007]: each node initially has a unique label, and at each iteration a node takes the most frequent label of its neighbourhood, until labels converge to a stable assignment. The ordering is obtained with a *layered label propagation* (LLP), which consists in applying the community detection algorithm with a range of resolution parameters so that the ordering represents different scales of communities.

3.3.6 Graph partitioning

↑ back to Table 3.1 ↑

Graph partitioning consists in cutting the graph into a number of subgraphs such that there are few connections between the subgraphs. Graph partitions are particularly useful to process the graph with parallel or distributed systems, as each unit can process one subgraph with limited need for communication between units. Graph partitioning can be seen as a type of community detection, but a variety of specific techniques and applications exist, as reviewed by Çatalyürek et al. [2023].

Formally, a graph partition of size k is a set $\{V_1, \dots, V_k\}$ where $\forall i, j \in [\![1, k]\!], V_i \subseteq V, V_i \neq \emptyset, V_i \cap V_j = \emptyset$ and $\bigcup_i V_i = V$. In general, two types of properties are sought in such a partition. First, the subgraphs should have approximately the same number of nodes, a property called *balancing* whose strictest form corresponds to $|V_1| = \cdots = |V_k| = \frac{n}{k}$. Second, there should be few edges between the subgraphs. The main metric for this property is the size of the edge cut, which is the number of edges that cross from one subgraph to another: $\sum_{i \neq j} |E \cap V_i \times V_j|$. Another metric is the *communication volume* defined by Bourse et al. [2014]: it is the sum over the nodes of the number of subgraphs with which they have an edge, or $\sum_i \sum_{u \in V_i} |\{j \in [\![1, k]\!], \exists v \in N_u \cap V_j\}|$; a low value is desirable as it means that, on average, the nodes have most of their neighbours in the same few subgraphs.

Node orderings and graph partitions are tightly related concepts: a partition can be obtained from an ordering by cutting it into regular blocks; conversely, an ordering can be obtained from a partition by sequentially taking the nodes of each subgraph. The underlying mechanism is grouping, as the desired properties apply on groups of nodes (the subgraphs of the partition) rather than on individual nodes and neighbourhoods.

Metis orderings. Karypis and Kumar [1998] follow a grouping mechanism and propose a general method in three phases to partition a large graph using graph reduction. The coarsening phase consists in progressively merging nodes together to reduce the size of the graph while keeping some of its properties. Then, the resulting smaller graph undergoes a partitioning process. Finally, the uncoarsening phase restores all the initial nodes into the corresponding subgraph. Several algorithms exist for each phase, but the authors report after extensive experiments that the choice of algorithm does not significantly impact the resulting cuts: rather, the performance of this scheme is due to the multiple scales of the graph that are taken into account.

Personalised Pagerank. The personalised Pagerank is a variation of Pagerank (3.3.1) that concerns a walk that starts on a particular node. The vector of scores p for each node results from the following matrix equation: $p = \alpha p W + (1-\alpha)v$, where W is the probability transition matrix of the random walk and and v is the vector that identifies the starting

node. Andersen and Chung [2007] rank the nodes according to their score in p, and prove that there can be a sharp drop in the value of p. In that case, the nodes of high p score form a subgraph of low conductance, which is a measure of the ratio between inner edges and outer edges. A graph partition can be obtained by repeating this selection over the other nodes. Other community detection methods may also be used in a similar way.

LDG ordering. Stanton and Kliot [2012] propose the *linear deterministic greedy* (LDG) heuristic, a streaming algorithm that computes a partition during a single pass over the nodes. When a node arrives, it is placed into the subgraph where most of its neighbours belong. To prevent imbalance between the subgraphs, a multiplicative penalty is applied to subgraphs that contain many nodes. This procedure takes linear time and terminates with a balanced partition, and it is reported to be resilient to the stream order (the order in which nodes arrive).

Fennel ordering. Addressing the one-pass streaming model as well, Tsourakakis et al. [2014] propose a generalisation of LDG that optimises a trade-off between node balancing and partition quality. On the one hand, the subgraphs may have a slightly different number of nodes if it improves the partition quality. On the other hand, this quality can be measured not only by the edge-cut, but also for instance by the modularity [Newman, 2006], a measure of relative connectivity that is widely use for community detection.

Sheep ordering. Margo and Seltzer [2015] build the *Sheep* partitioning method as a heuristic to minimise the communication volume over large distributed graphs. The algorithm uses a graph reduction method: it first reduces the graph to an elimination tree, defined as a rooted tree such that if two nodes are neighbours in the initial graph, then one is an ancestor of the other in the tree. A partition process is applied on the tree and then extended to the other edges of the initial graph. The partition depends on the initial ordering of the nodes, and the authors show that an ordering minimising the depth of the elimination tree is optimal. They note that a proxy for such an ordering is one that attacks the graph efficiently (see Section 3.3.1), and they choose the degree ordering for its scalability.

Vebo ordering. While classical graph partitioning aims at subgraphs that have a balanced number of nodes, Sun et al. [2018] point out that the number of edges could be balanced as well. Indeed, if subgraphs are processed in parallel, the total execution time will benefit from subgraphs of even size, both in terms of nodes and edges. For the authors, this load-balancing constraint is not straightforward in real-world networks: with their skewed degree distribution, nodes can have a strongly different number of edges, which means that node balancing does not guarantee edge balancing. The Vebo partitioning strategy reads nodes in decreasing order of their degree and places them in the subgraph that best realises the node and edge balance constraint. It follows a grouping mechanism and can be considered as a streaming algorithm with several passes.

3.3.7 Other applications

↑ back to Table 3.1 ↑

Balanced ordering. Biedl et al. [2005] introduce the problem of finding an ordering where nodes have the same number of predecessors and successors, which they call a *balanced ordering* (to be distinguished from the balancing property of graph partitions

mentioned in Section 3.3.6). This is a precedence problem that corresponds to the minimisation of the objective function $\sum_{u \in V} |d_u^+ - d_u^-|$. On top of its applications in graph drawing, this problem is interesting for its theoretical aspect: Biedl et al. present an insightful proof of NP-hardness and find approximation algorithms; Kara et al. [2007] study the complexity of variations of the initial problem for different classes of graphs.

Revorder. The *referenced vertex ordering* consists in finding an ordering where each node has at least a certain number U of predecessors, or references. The question that arises is the minimum number K of nodes that break this rule. Omer and Mucherino [2021] present hardness results for different values of U and K, an exact branch-and-bound algorithm as well as greedy approximations, and experimental analysis. The underlying mechanism is precedence, and the authors express the problem as the optimisation of an objective function.

3.4 Algorithmic problems with ordering formalism

↑ back to Table 3.1 ↑

On top of their specific domains of applications, node orderings are sometimes subject to their own algorithmic problem. In fact, a variety of classical graph problems can be reformulated as an optimisation over the set of node orderings. Bodlaender et al. [2012] propose a unification that consists in finding an ordering π minimising an objective function of the form $\sum_{u \in V} f(G, u, \Pi_u)$ or $\max_{u \in V} f(G, u, \Pi_u)$, where f is any function that can be computed in polynomial time and Π_u is the set of nodes that appear before u in π. An exponential algorithm for the Travelling Salesperson Problem (TSP) is able to solve these generic problems using polynomial space, which gives a direct way to solve any problem that can be expressed with these formulas. In particular, the authors address the MinLinA problem (3.3.4) and its directed version, as well as the problems of tree-width, path-width, cut-width, fill-in, and feedback arc set that we do not define here.

Notably, the colouring problem can be understood as a node ordering optimisation. Colouring a graph consists in giving to each node u a colour $c(u)$ while ensuring that neighbours have distinct colours. The colouring problem consists in finding a graph colouring with as few colours as possible, or to minimise $|c(V)|$. This problem is one of the classical NP-hard problems of Karp [1972]. A famous greedy algorithm is often used to compute a colouring that is not necessarily minimum: using integers as colours, the algorithm streams nodes in a given order, and colours each of them with the lowest colour that is not taken by its neighbours. The choice of ordering matters, and Asik et al. [2020] identify closeness centrality as a better strategy than degree at distance one, two or three, Pagerank, clustering coefficient, or random. Mathematically, there even exists an ordering that leads to optimal colouring: if nodes are sorted according to their colour in a minimum colouring c, then the greedy algorithm is able to find the same colouring. In other words, the problem of minimum colouring can be considered as an ordering problem.

The vertex cover problem, and its complementary independent set problem, also admit a formulation with orderings. A vertex cover is a subset of nodes that covers all the edges of the graph, which means that each edge has at least one node in the subset. The minimum vertex cover problem consists in finding a smallest subset with this property. This NP-hard problem is studied in Chapter 6, where we present a method to certify the quality of heuristics in practice. To reformulate it as an ordering problem, let us consider a minimal vertex cover $C \subseteq V$. We define an ordering π in which nodes of C come before the other nodes: $\forall u \in C, v \in V \setminus C,\ \pi_u < \pi_v$. Suppose that the out-degree of

$u \in C$ is zero, then all of its neighbours are also in C, so C is not minimal: necessarily $d_u^+ > 0$. On the other hand, all the neighbours of a node $v \in V \setminus C$ have to be in C, so $d_v^+ = 0$. Thus, the size of the vertex cover is precisely the number of nodes that have a positive out-degree, hence this node ordering formulation of the minimum vertex cover problem: given an undirected graph $G = (V, E)$, find the node ordering π that minimises $|\{u \in V \text{ with } d_u^+ > 0\}|$.

The problem of ordering recovery consists in inferring the order in which nodes appeared during the growth of the network. Turowski et al. [2020] study a model of growth called duplication-divergence: a newly created node copies the adjacency of an existing node and applies random deletion and addition of edges. Assuming such a growth model, they compute an approximate solution to the order recovery problem.

3.5 Overhead considerations

↑ back to Table 3.1 ↑

As we have seen, node orderings are used to improve algorithms in a variety of problems. One of their most interesting capability is to reduce the execution time of algorithms. Yet, as soon as time measurements are made, the question of trade-off occurs: if using an ordering spares some time in the execution, how long can we spend in computing the ordering? We call *overhead* the pre-processing time spent to compute the ordering.

Distinct scenarios lead to different answers. First of all, one may consider the ordering procedure as part of the general algorithm. In this situation, the gain of time caused by the ordering has to be higher than the overhead. It is usually the case for orderings of low complexity such as the degree ordering in $\mathcal{O}(n)$. When more complex orderings are involved, this constraint is only possible for algorithms that have a high complexity, such as clique enumeration or pairwise shortest paths computation. Overhead considerations also raise the question of the parallelism of node orderings: if the main algorithm can run in parallel on many processors, it is important that the ordering phase can also run in parallel. For instance, parallel versions have been developed for RCM ordering by Karantasis et al. [2014] and for core ordering by Shi et al. [2021] in order to reduce the overhead in the case of parallel execution.

At the other end of the spectrum, one may consider that the overhead does not matter. Indeed, it is possible to compute an ideal ordering once, and then to relabel the nodes to store the ordering in the graph file. If this file is distributed to other users or processed many times, the overhead can be amortised. This is for instance the case in large database systems, where queries have to be answered in a short time but an ordering algorithm can run in the background. However, this situation can be paradoxical when the ordering comes from the optimisation of an objective function that is as hard as the initial problem: in some reformulated graph problems of Section 3.4, there exists an ordering that allows a greedy linear algorithm to find an optimum solution. Thus, finding the ordering is as hard as solving the problem. Moreover, the resulting ordering only solves this specific problem and cannot necessarily be used for other tasks.

Most researchers imagine an intermediate situation where some overhead is tolerated as long as the whole process is still scalable. The overhead may also be acceptable when the benefit of orderings is not an acceleration: it can be a higher compression rate or a smaller cut, in which case there is no clear rule as to how long to spend on the ordering phase.

Chapter 4

Orderings for faster execution

Reducing the cache-miss rate of graph algorithms

Summary

As explained in Chapter 2, the role of a cache system is to store pieces of data close to the processor so that it can access it faster than if it were only available in main memory. When an algorithm requires a piece of data at a point of its execution, it benefits from this acceleration if the data is contained in a cache line; otherwise, it has to wait for it to be retrieved in main memory, which is called a *cache-miss*. Avoiding cache-misses has repercussions over the total execution time of the algorithm, as they sometimes amount for more than half of the computation time. In order to reduce cache-misses for generic graph algorithms, Wei et al. [2016] propose to reorder the nodes so that two nodes that will end up on close in memory are also closely linked in the graph. They propose a new ordering called Gorder that approximates this objective with a greedy procedure. This chapter is a replication of their work with additional results and discussions on their contribution. We implement ten existing ordering methods and measure the execution time of nine standard graph algorithms on large real-world datasets. Monitoring cache performances allows us to confirm that variations in the execution time are due to differences in the proportion of cache-misses.

As the authors of the initial paper documented, Gorder leads to the fastest execution in most cases. Yet, our results show that simpler orderings based on graph traversal are almost as efficient in terms of cache-miss reduction, even though they are much faster to compute. Our replication highlights that the problem of cache optimisation is subject to concerns of overhead which, as explained in Section 3.5, require to think of ordering methods as a trade-off between initial time investment and acceleration of subsequent executions.

Contributions

* Replicate all the experiments of the paper of Wei, Yu, Lu, and Lin [2016].
* Implement all the orderings and algorithms in a unified open-source code.
  https://github.com/lecfab/rescience-gorder
* Compare the benefits of orderings for a variety of algorithms and networks.
* Provide additional discussion on the performance of different orderings.

Publication

* [*Replication*] *Speedup graph processing by graph ordering*
  Lécuyer, Danisch, and Tabourier [2021], ReScience.

4.1 Introduction

In graph algorithmics, various procedures use the same few atomic operations. Among the operations described in Section 2.1.3 to motivate the choice of graph data structures, neighbourhood listing is particularly important: accessing the neighbours of a given node is key to problems such as computing shortest paths, finding connected components, detecting communities etc.

Making this type of elementary operation faster would improve such algorithms without having to modify their implementation. That effect can be obtained by leveraging the cache architecture of computers. Figure 4.1 illustrates the different layers of memory of a standard computer, and more information is given in Section 2.3. The graph data is stored in main memory, but pieces of it are also duplicated in different levels of cache so that the processor can access it faster. When the data is not available in cache, a cache-miss occurs: the processor has to fetch it in main memory, which is up to twenty times slower depending on the machine. Ailamaki et al. [1999] and Cieslewicz and Ross [2008] show that the cache stall, defined as the time lost in cache-misses, represents a significant share of the computation time, which gives incentive to reduce it.

Reducing the cache-misses is an option of choice to accelerate elementary operations. Specifically, two variables that are often accessed together by algorithms should be stored side-by-side in memory so that they are duplicated together on a cache line. In a graph stored as adjacency lists, it means reordering the nodes so that neighbours have close-enough indices.

This chapter replicates the paper of Wei, Yu, Lu, and Lin [2016] which introduces Gorder, a new procedure to order the nodes of a graph to reduce cache-misses, and compares it to other standard orderings using typical algorithms and datasets as benchmarks.

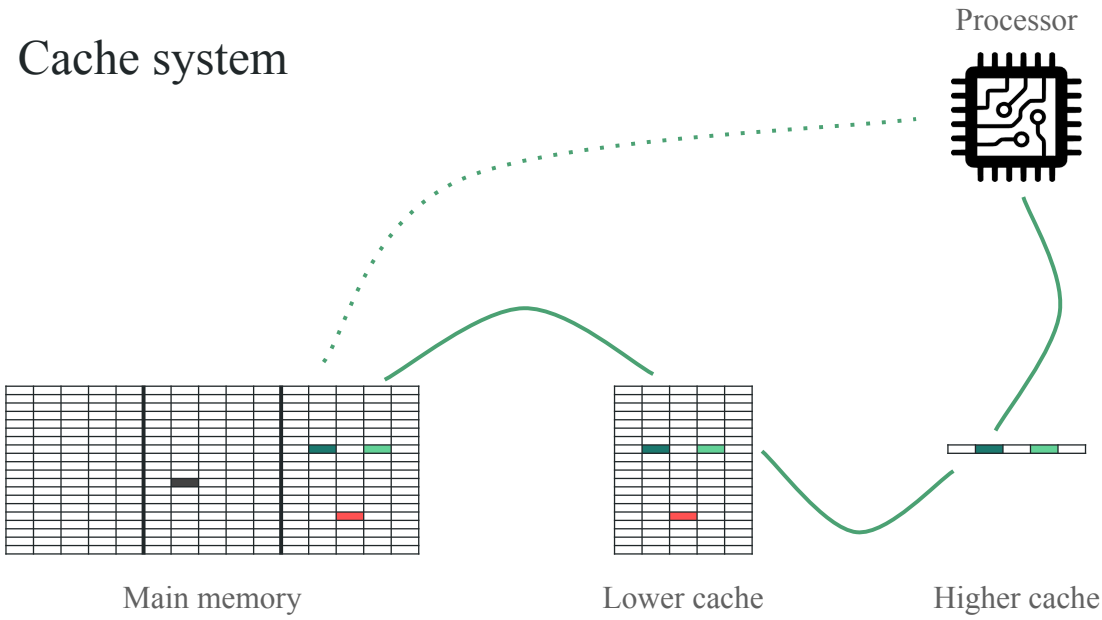
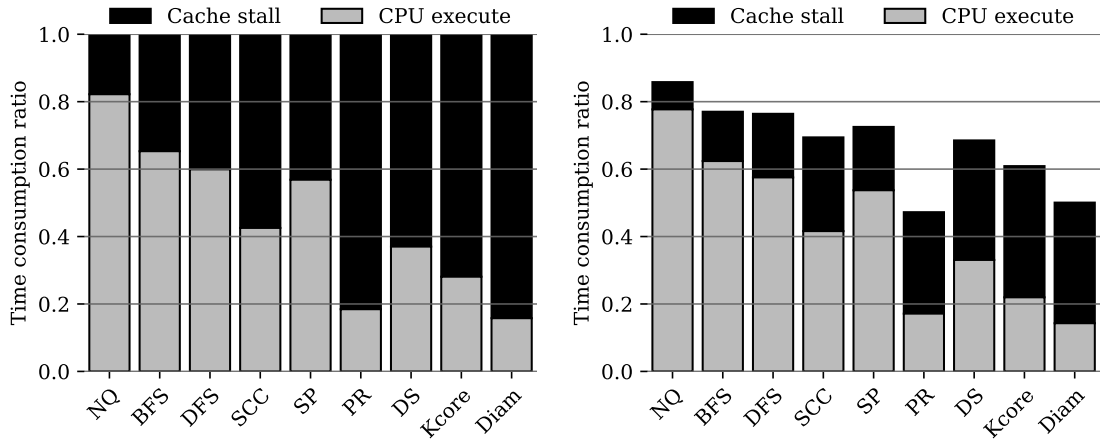


Figure 4.1: **Representation of a cache system.** Main memory (RAM) is large and cannot be stored close to the processor; duplicates of data are stored in smaller, closer memory units called cache levels. The execution of a program is faster when the data required by the processor is in the cache.

Testing a variety of graph algorithms favours orderings that affect elementary operations that are used in most algorithms, as opposed to orderings that would be tailored for one specific algorithm. The authors of the original paper claim an improvement of 10 to 50% in runtime, due to lower cache-miss rate.

We were able to replicate most of the experiments and confirm that ordering the nodes according to Gorder makes the implementations 10 to 50% faster than without reordering. Section 4.2 presents the algorithms, orderings and datasets as well as issues faced during the replication. Our results are presented in Section 4.3 and are compared to the original results. Finally, Section 4.4 discusses the relevance of such an ordering compared to simpler ordering methods that offer satisfactory performances.



(a) Original ordering

(b) Gorder

Figure 4.2: **CPU execution and cache stall.** Original ordering and Gorder are compared for all algorithms on the *sdarc* dataset. Grey bars are time spent on CPU operations, black bars represent time spent waiting for data retrieval. Times are normalised by the total runtimes with the original ordering. Figure (a) shows the share of CPU execution and cache stall for the original ordering, figure (b) shows them when the network has been reordered following the Gorder procedure. While both need about the same CPU time, the latter is significantly faster due to cache stall reduction. Compare to Figure 1 in Wei et al. [2016].

4.2 Method

The original study of Wei et al. [2016] was motivated by the observation that cache stall can take up to 70% of the whole computation time, which is supported by the observations reported in Figure 4.2. While researchers have addressed the issue for specific algorithms, such as Then et al. [2014] for breadth-first search, Wei et al. propose a more general approach that aims at optimising the data organisation to improve cache performance, regardless of the algorithm or hardware specifications.

Gorder clusters nodes that are likely to be accessed simultaneously by any graph algorithm. More precisely, let us consider a directed graph $G = (V, E)$ with $n = |V|$ nodes and $m = |E|$ edges. The proximity of two nodes u and v is measured by a score $S(u, v)$ that increases if they are neighbours and if they share many common predecessors. The total score F is the sum of $S(u, v)$ for all nodes u and v that have close indices, where the

closeness is defined by a parameter called the window size w. The score F naturally depends on the node ordering π, and Gorder aims at finding an ordering that maximises F. A more formal definition of this optimisation problem is given in Section 4.2.3.

The authors prove that finding the optimal ordering π is a NP-hard problem and propose a heuristic with a theoretical approximation bound. They also present algorithmic improvements that reduce the time complexity of the heuristic. Finally, they run extensive experiments to compare Gorder to other standard orderings.

Although the theoretical results of the original paper are important to explain the efficiency of Gorder, we only focus here on the algorithms and experiments. Wei et al. provide an extensive analysis by comparing the runtimes for nine typical algorithms on eight large datasets with nine possible orderings. The current section describes them all and exposes the issues that we faced during our replication. All the codes and instructions for this purpose can be found in our repository.

4.2.1 Algorithms

In the original paper, the authors selected typical graph algorithms to test the different orderings. However, their implementation details were not fully documented and they were unable to provide additional information upon request. As a result, we had to make implementation choices that we detail below to ensure transparency and reproducibility. Here are the nine algorithms that are executed by Wei et al. and in our replication.

Neighbour query (NQ). Listing the neighbours of a given node is a standard elementary operation in graph algorithmics. As defined by Wei et al. [2016], this operation must "*access the out-neighbours*" of each node. To assess whether the cache benefits from a favourable ordering, we need to ensure that each neighbourhood is duplicated in cache at some point. To do so, we choose to execute an arbitrary operation over the set of neighbours of each node: we compute for each node u the sum of degrees of its neighbours $q_u = \sum_{v \in N_u} d_v$.

Breadth and Depth-first search (BFS, DFS). BFS and DFS are standard graph traversal algorithms [Cormen et al., 2009] that we adapted to the data structure detailed in Section 4.2.2. Note that neighbours are selected in lexicographic order defined by the original ordering of the network.

Strongly connected components (SCC). A strongly connected component is a maximal subset of nodes such that there exists a path from any of its nodes to any other of its nodes. Finding the SCCs of a graph is a classical problem for which a well-known algorithm was proposed by Tarjan [1972]. Our implementation of this algorithm is based on DFS.

Shortest paths (SP). As in the original paper, we use the Bellman-Ford algorithm [Cormen et al., 2009] to compute the minimum distance from a source node to any other node. The time complexity after simple optimisations is in $O(\Delta \cdot m)$ where Δ is the diameter of the graph and m is the number of edges. As real-world networks are known to have relatively small diameters ($\Delta \ll n$), this algorithm works on the large networks presented in Section 4.2.2. Note that for unweighted graphs, shortest paths can be computed in linear time and space using a BFS, but we keep the algorithm suggested by Wei et al. for comparison purposes.

Page rank (PR). Pagerank is the method presented by Page et al. [1999] to rank webpages. Based on random walks, it gives a score to each node according to its importance in the network structure. As computing the exact Pagerank is not a scalable problem, Wei et al. use a heuristic called the power iteration method. We implement this heuristic and execute it with 100 iterations and a damping factor set to $\alpha = 0.85$, which is a usual configuration.

Dominating set (DS). A dominating set is a subset of nodes such that every node of the graph either belongs to the subset or has a neighbour in it. The implementation is not described in the original paper so we use a greedy approximation [Cormen et al., 2009] made of two steps. First, we select the node with the most uncovered neighbours and add it to the dominating set. Second, this node and all its neighbours are removed from the graph because they are now covered. The two steps are then repeated among the remaining nodes.

Core decomposition (Kcore). Described in Chapter 3, the graph peeling algorithm of Batagelj and Zaveršnik [2003] consists in recursively removing the node of smallest degree until only a core of well-connected nodes remains. We use a priority queue structure implemented as a binary tree to keep track of the degrees, leading to a quasi-linear time complexity.

Diameter (Diam). The diameter of the graph is the longest distance between two nodes. Efficient approximations with theoretical bounds exist, such as the ones based on BFS presented by [Corneil et al., 2003]. Instead, Wei et al. [2016] run 5000 times the shortest paths algorithm SP from a random node, and output the highest distance obtained. Even though this method is not the most accurate and efficient to compute the diameter, it is still interesting in this study where the goal is to compare the performances of different orderings in terms of cache-misses.

4.2.2 Datasets and data structure

Eight real-world datasets are used as benchmarks in the work of Wei et al.. Their basic features are reported in Table 4.1. As per usual with real-world graphs, these graphs have specific properties described in Section 2.2.1: sparsity ($m \ll n^2$), small diameter, skewed degree distribution, etc. As shown in Table 4.1, their sizes range from 1.6 million nodes and 30 million edges to almost 100 million nodes and two billion edges. In order to facilitate further experiments, we attach the *epinion* dataset to the repository, a smaller network on which our code can be tested quickly.

These networks are the same as in the original paper, and the table provides the URLs where they can be downloaded. Wei et al. [2016] selected two main categories of real-world networks: online social platforms, where a node is a user and a directed edge represents a social interaction, and web graphs, where a node is a web page and a directed edge is a hyperlink.

The datasets are directed graphs given as lists of edges. In order to store large graphs in main memory, an efficient data structure is needed. Libraries exist for that purpose, but we develop our own light structure to have better control over the implementation of the algorithms. A list of edges does not provide quick access to the list of neighbours of a given node, which is the crucial operation for most of the above graph algorithms. The data is therefore converted into a adjacency lists, where a node points to the list of its

Dataset	Size (Go)	Nodes (10^6)	Edges (10^6)	Source	Category
pokec	0.4	1.63	30.6	SNAP[1]	Social
flickr	0.4	2.30	33.1	Konect[2]	Social
livejournal	1.0	4.85	69.0	SNAP[1]	Social
wiki	6.7	13.6	437	Konect[2]	Web
gplus	7.3	28.9	463	Gong[3]	Social
pldarc	10	42.9	623	WDC[4]	Web
twitter	26	61.6	1470	Kaist[5]	Social
sdarc	34	94.9	1940	WDC[4]	Web
epinion (added)	0.005	0.0759	0.509	SNAP[1]	Social

Table 4.1: **General features of the datasets used in the experiments.** The data can be found in the following websites:
[1]Stanford Network Analysis Project: http://snap.stanford.edu/data/
[2]Koblenz Network Collection: http://konect.cc/networks/flickr-growth/ and http://konect.cc/networks/wikipedia_link_en/
[3]Gong Research Group: http://gonglab.pratt.duke.edu/google-dataset
[4]Web Data Commons: http://webdatacommons.org/hyperlinkgraph/2012-08/download.html
[5]Kaist Advanced Networking Laboratory: http://an.kaist.ac.kr/traces/WWW2010.html

neighbours. For efficient storage, we use the compressed sparse row format described in Section 2.1.3.

4.2.3 Orderings

We list below the different ordering methods considered in the original study and used as a benchmark for comparison with Gorder. In our replication, we added, modified or removed some of them for practical reasons. An ordering is defined as a permutation π where π_u is the index of node u. See Chapter 3 for more details and context on these and other ordering methods.

Original. Datasets are collected in a way that is not random but is rarely reported. The original ordering is the one that is contained in the network file when we download it.

Random (added). The original paper did not include a random ordering. We include it as a non-favourable benchmark for comparison to all other orderings. Our implementation obtains a random ordering by shuffling the indices of nodes.

MinLinA and MinLogA. The *minimum linear arrangement* (MinLinA) and the *minimum logarithmic arrangement* (MinLogA) problems have been presented in Chapter 3. The goal is to find an ordering π of the nodes that minimises a given "energy" function. The energy $\mathcal{E}$ is computed over the set E of edges in the following way:

$$\mathcal{E}_{MinLinA} = \sum_{(u,v)\in E} |\pi_u - \pi_v| \qquad \text{and} \qquad \mathcal{E}_{MinLogA} = \sum_{(u,v)\in E} \log|\pi_u - \pi_v|$$

As both exact minimisations are NP-hard, a heuristic method is necessary. Wei et al. [2016] use the meta-heuristic called simulated annealing: random swaps are achieved

to decrease the energy $\mathcal{E}$ with some tolerance for swaps that increase it, while the temperature goes down and reduces the tolerance for such bad swaps. The parameters of simulated annealing are known to be hard to tune, and our implementation has two of them: the number of steps S (how many random swaps are performed before stopping) and the standard energy k (what typical tolerance is given to bad swaps). The temperature T decreases linearly so that, at step s, $T(s) = 1 - s/S$. At each step, two nodes are picked at random. Swapping their indices in π leads to a variation δ of the total energy $\mathcal{E}$. If δ is negative, the swap is registered and the energy decreases. Otherwise, it is registered with a probability p, inspired by statistical physics:

$$p(\delta, T) = \exp\left(-\frac{\delta}{k \cdot T}\right)$$

In Figure 4.3 we test a wide range of values of S and k on *epinion*. While we cover a significant fraction of the parameter space, we are not able to find a combination of S and k that outperforms a simple local search ($k = 0, p = 0$), where only the favourable swaps are accepted. In our next experiments, we set $S = m$ and $k = m/n$.

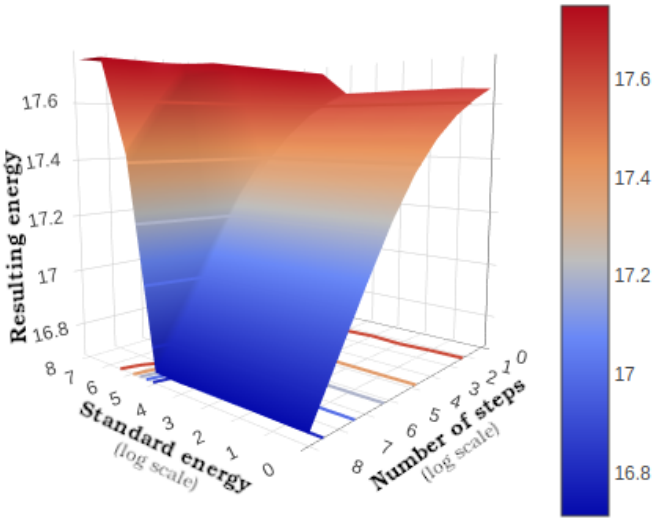


Figure 4.3: **Tuning simulated annealing for MinLinA.** The colour and the z-axis represent the energy of the best permutation π obtained for various values of the parameters (standard energy k and number of steps S) on *epinion*. S ranges from n to $m \log n$ (logarithmic scale), which we consider to be the maximal acceptable time for the heuristic implementation. Standard energy k ranges from $1/(mn)$ to mn (logarithmic scale). We observe that a) the higher S, the lower the resulting energy, b) when k is high, the resulting energy is at a maximum: all swaps are accepted regardless of their quality which results in a random arrangement, c) any low value of k yields the same energy as $k = 0$ which corresponds to a local search, in particular $k = m/n$ which we use afterwards.

RCM. The Reverse Cuthill and McKee [1969] ordering is a BFS ordering where nodes of small degree have priority. It is meant to find an arrangement π that reduces the bandwidth of a sparse graph, given by $\max_{(u,v)\in E} |\pi_u - \pi_v|$ with π_u the index of node u.

Degsort. As proposed in the original paper, we sort the nodes in descending order of in-degree, breaking ties with the lexicographic ordering of the node indices.

Chdfs. Wei et al. [2016] mention the *children-depth first search traversal*, which we assume to be a usual DFS algorithm. In our implementation, the first node is chosen at random, then the selection of children is made following the original ordering of node indices.

Slashburn (simplified). Slashburn is an iterative process that separates hubs (high-degree nodes) from low-degree nodes connected to hubs. Our implementation creates an ordering by iterating over an array of size n, initially empty. Each iteration divides the array in parts A, B and C. Part A takes only one node, selected at random among those with highest degree. Nodes that have no neighbour go to part C. Then this hub and these isolated nodes are removed from the graph, which creates new isolated nodes, and degrees are updated. Part B is filled by the next iteration until no node remains.

There are two differences between our implementation and the original Slashburn algorithm introduced by Lim et al. [2014]. First, the official version fills part C with disconnected components instead of isolated nodes. Second, it puts r hubs in part A, and r is a parameter that can be modified. As no precise information was given by Wei et al. [2016] on their choice of parameters and algorithms to extract the main connected component, we implement the simpler version described above instead.

LDG. The Linear Deterministic Greedy partitioning of Stanton and Kliot [2012] creates $\frac{n}{k}$ bins of size k and streams the nodes to put them in their preferred bin. Their preference is defined by the number of their neighbours that belong to a given bin. Larger bins are penalised in order to ensure that at the end of the process all bins have equal sizes (up to one element). More precisely, a node u with a neighbourhood N_u is placed in a bin that achieves

$$\arg\max_{\text{bin } B} \Big(1 + |N_u \cap B|\Big) \times \Big(1 - \frac{|B|}{k}\Big)$$

At the end of the process, each bin contains about k nodes. Wei et al. [2016] choose $k = 64$ so that a bin can fit on a cache line. Indeed, common contemporary processors have L1 caches of a few dozen kilobytes (32kB in our case) made of lines of 64 bytes each.

Metis (removed). Metis is a powerful and extensive tool for graph partitioning. A c++ implementation is available [1] but it is not suitable for large graphs: the original paper could only test it on the three smallest networks because of its excessive memory consumption. Since this ordering does not scale, we do not test it in our experiments.

Gorder. Gorder is the ordering method proposed by Wei, Yu, Lu, and Lin. They give a precise description of its implementation and release an open-source c++ implementation [2]. As mentioned at the beginning of Section 4.2, the authors define the quality function F of an arrangement π by:

$$F(\pi) = \sum_{0<\pi_u-\pi_v\leq w} S(u,v) = \sum_{0<\pi_u-\pi_v\leq w} \Big(S_s(u,v) + S_n(u,v)\Big)$$

where w is the window size; $S_s(u,v)$ is the number of times u and v coexist in sibling relationships or their number of common predecessors; $S_n(u,v)$ is the number of times they are in a neighbour relationship, which is either 0, 1 or 2 since both directed edges (u,v) and (v,u) may exist.

[1] http://glaros.dtc.umn.edu/gkhome/metis/metis/overview
[2] https://github.com/datourat/Gorder

The greedy algorithm presented by Wei et al. creates the ordering π by recursively inserting the node that has the highest proximity to nodes presently within the window. Storing the proximity scores S requires a complex structure called unit heap, made of a linked list and pointers to different positions. We took the functions provided in the original code and adapted them to our data structure.

To choose the parameter w for the window size, Wei et al. [2016, Figure 8] create versions of Gorder for w ranging from 1 to 8. For each version of w, they run the PR algorithm on the *flickr* dataset. The idea is to check which parameter allows for the fastest runtimes in this particular example and to generalise it to other algorithms and datasets. The fastest runtime is obtained with $w = 5$, so they use this value in their experiments. However, there is only a small relative variation of runtime (3%) between the 8 versions. In Figure 4.4 we compare a wider range of window sizes, as w could in theory be anything between 1 and n. We find that setting w between 64 and 2048 gives a further 3% speedup compared to $w = 5$. Note that our next experiments use $w = 5$ to ensure strict replication.

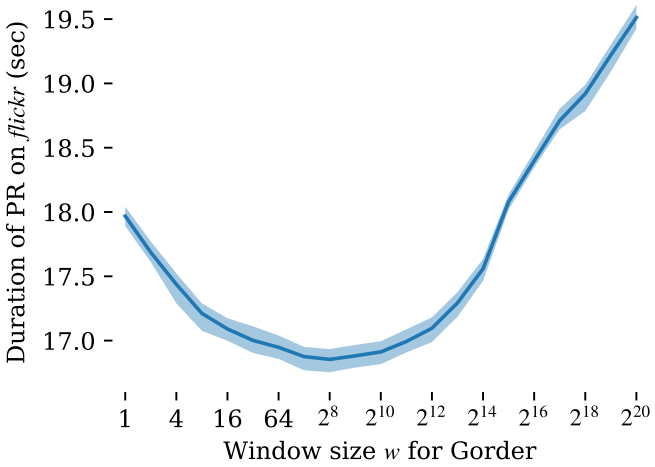


Figure 4.4: **Tuning window size.** Versions of Gorder obtained for window sizes ranging from $w = 1$ to $w = 2^{20} \simeq 10^6$ are tested in Pagerank over *flickr* (that has $n \simeq 2 \cdot 10^6$ nodes). Median execution time and 90% confidence interval are shown for 100 repetitions. The plateau from $w = 64$ to $w = 2^{11} = 2048$ gives better results than $w = 5$. This figure can be compared to Figure 8 in the paper of Wei et al. [2016]; note that the absolute runtimes are different because of hardware differences.

For the purpose of replication, we also use $w = 5$ in the next experiments. Two other factors make the choice of a small w relevant. First, the computation of Gorder is faster when the window is narrow, because a candidate node has to compute its proximity score S with all the nodes of the window. Second, the authors of the original paper show that their heuristic is a $\frac{1}{2w}$-approximation of the optimal score: reducing w makes this bound tighter.

4.3 Results

4.3.1 Implementation hardware

To deal with bigger datasets and avoid the fluctuations of time measurements described in Section 2.3, we run the experiments on an isolated cluster (SGI UV2000 Intel

Xeon E5-4650L @2.6 GHz, 128GB RAM). Each processor has three levels of cache of respective size 32kB, 256kB and 20MB. The hardware of Wei et al. has similar cache and RAM storage but higher clock frequency, which can explain the differences in runtime (in addition to programming techniques and optimisation). Note however that these differences should not modify the relative performance of different orderings.

4.3.2 Ordering time

Computing an ordering on a large network can be a long process, and some of the ordering methods have limited scalability. As mentioned above, Metis has been removed from the experiments for this reason. Table 4.2 reports the duration of the computation of each ordering. For datasets that have less than a hundred million edges, they can all be computed in a couple of minutes at most, with DegSort and ChDFS orderings requiring less than a second.

When the number of edges rises however, the computation takes hours for MinLinA, MinLogA, and Gorder. In the case of MinLinA and MinLogA, the number of steps is chosen arbitrarily as described in Section 4.2.3. The process could thus be interrupted earlier, at the cost of a less efficient resulting ordering. As for Gorder, it does not scale linearly and its processing speed decreases as the number of edges increases. For example, the algorithm processes 380k edges per second for *pokec* but only 60k edges per second for *sdarc*, which requires a computation time of almost 9 hours. To speed up the process, a smaller window size can be used, but this results in a slightly less efficient ordering, as shown in Figure 4.4.

	pokec	*flickr*	*livejournal*	*wiki*	*gplus*	*pldarc*	*twitter*	*sdarc*
MinLinA	28	27	92	441	539	579	**2956**	**4884**
MinLogA	89	64	217	**2169**	1662	**2258**	**10245**	**17168**
RCM	3	5	10	60	49	63	158	406
DegSort	0.8	0.4	1	5	9	14	30	85
ChDFS	1	0.8	1	3	8	10	54	76
Slashburn	3	9	16	37	90	189	633	1066
LDG	6	7	13	68	101	144	673	798
Gorder w=5	79	110	118	988	**3324**	**8783**	**25475**	**32488**
Edges m	31M	33M	69M	437M	463M	623M	1.47G	1.94G

Table 4.2: **Graph ordering time.** We indicate the time to compute each ordering in seconds (in bold font when above 30 minutes). We also indicate the number of edges for each dataset to help evaluating the scalability of a method. Comparing to Table 9 of Wei et al. [2016] is possible for RCM, DegSort, ChDFS and Gorder because the implementations are alike: it shows that the hardware of Wei et al. is 2 to 5 times faster than ours. For the other orderings, the implementations are likely too different to be compared.

4.3.3 Running time

The main purpose of Wei et al. [2016] is to assess whether the node orderings listed lead to a faster execution of standard graph algorithms. For each combination of dataset, ordering and algorithm, we measure the execution time. The *speedup* is defined as the ratio between the execution time with a given ordering and the execution time with Gorder. Figure 4.5 shows the performances of all the orderings compared to Gorder. The results

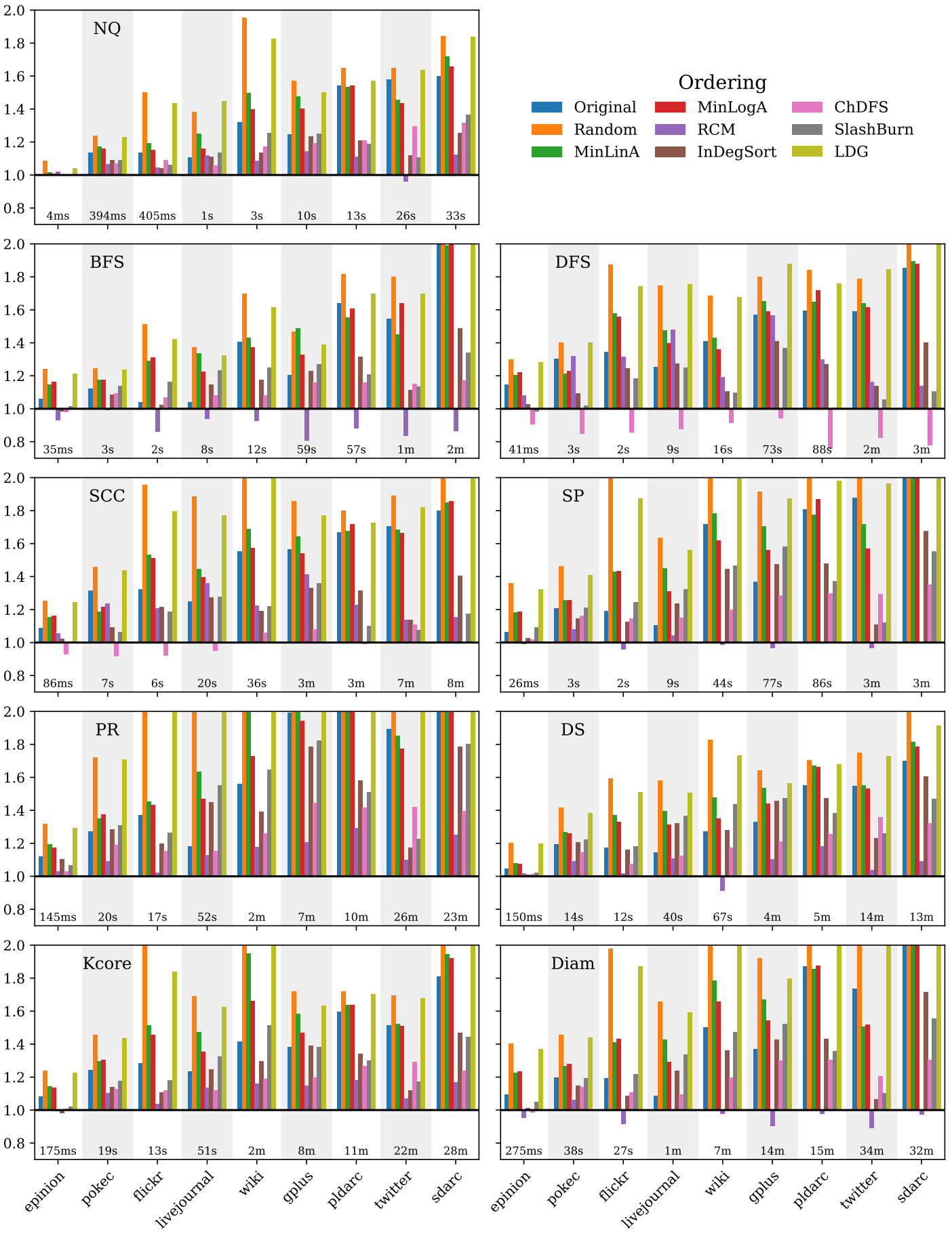


Figure 4.5: **Speedup of Gorder.** For each algorithm and each dataset, we display the absolute runtime for Gorder. Bars represent the relative time of all other orderings compared to this reference. For readability, the y-axis is cut above factor 2, but values go as high as 3.7. This figure can be compared to Figure 9 in Wei et al. [2016]. Another visualisation is show in supplementary figure 4.6.

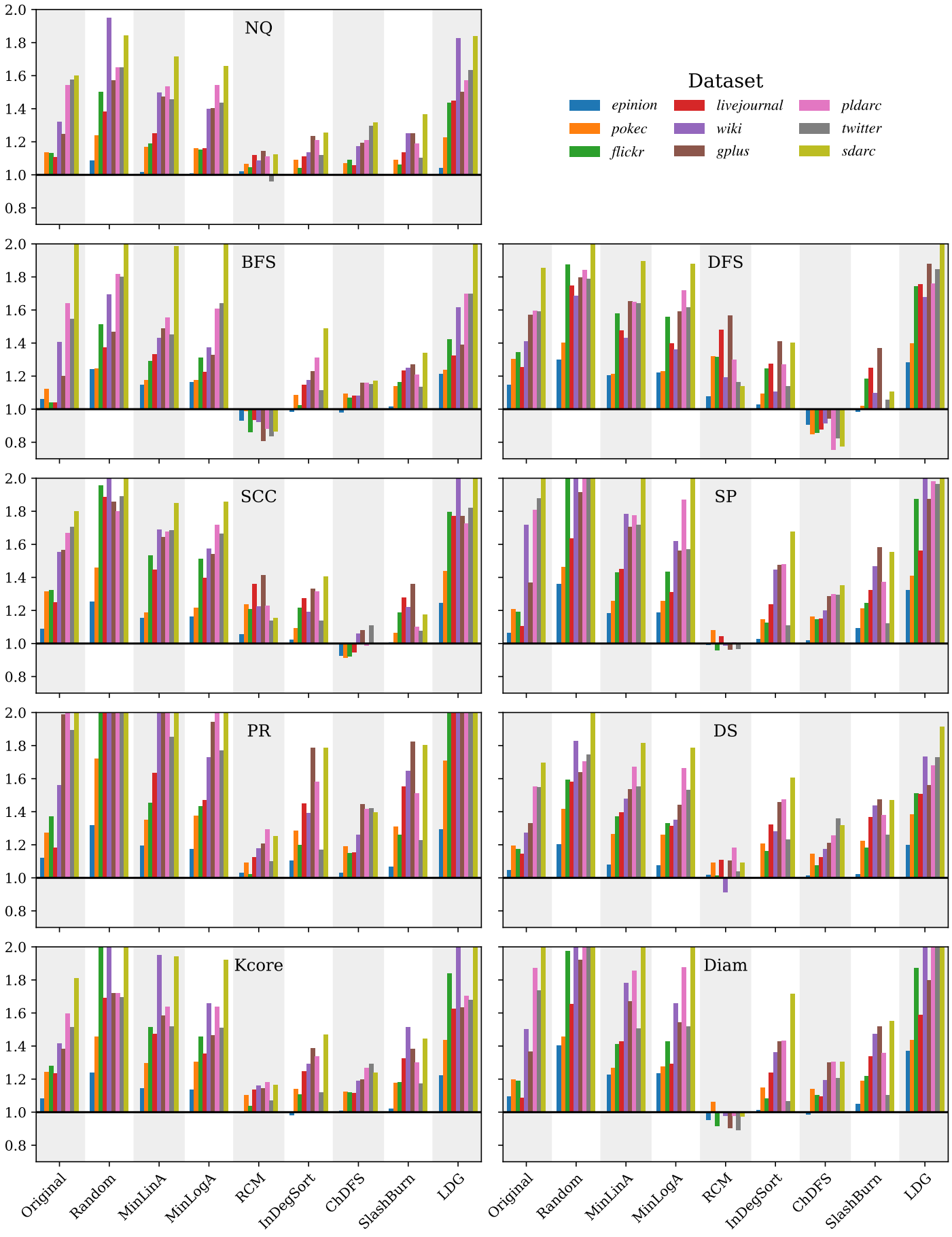


Figure 4.6: **Speedup of Gorder grouped by ordering.** For each algorithm and each ordering, bars represent the relative duration on each dataset compared to Gorder, taken as a reference. For readability, the y-axis is cut above factor 2. This figure displays the same information as Figure 4.5 but groups bars by ordering instead of dataset.

are reported for each dataset and algorithm in the same way as Figure 9 of the original paper. Additionally, we propose in Figure 4.6 another visualisation of the same results but grouped by ordering instead of dataset, which emphasises the overall performance of each ordering method.

A first few observations can be made from the raw results of the experiments. Like Wei et al., we observe that Gorder almost always leads to the fastest execution times. The y-axis of Figure 4.5 is limited to a speedup of 2 for clarity, but the the values reach up to 2.5 compared to the original ordering (for Diameter on *sdarc*) and 3.7 compared to a random ordering (for Pagerank on *wiki*).

To help make sense of the results, we propose an aggregated visualisation in Figure 4.7, where each ordering is ranked according to its performance in comparison to the other orderings through the 81 series of experiments reported in Figure 4.5. It shows in particular that Gorder is the best ordering method in half of the experiments, and second-best in most other cases. Let us give a few comments on the performances of the different ordering methods under examination.

Random is the worst and original is medium

The experiments show that the default ordering of networks performs better than more elaborate methods such as MinLinA or LDG, that have a high computation overhead as shown in Table 4.2. This phenomenon was also observed by Wei et al.. This indicates that the way in which datasets are constructed tends to give close indices to nodes that are in the same neighbourhood, and that the original ordering is not a random ordering. In a web graph for instance, if webpages are listed alphabetically by URL, it is likely that two consecutive nodes have a hyperlink between them since they belong to the same website. The random ordering is always the worst performer except for 6 experiments where it is second-to-worst. It is not surprising as each of the other orderings tends to bring adjacent nodes together, which should improve the cache efficiency and the algorithm runtime.

Some elaborate orderings perform poorly

Note however that LDG performs only slightly better than random, and that it is almost always the slowest in the experiments of Wei et al. too. In a quarter of the experiments, the execution is more than twice as slow as with Gorder. These poor results lead to think that either its parameter (the size of bins $k = 64$) is not optimal, or that its quality function is not highly correlated to cache efficiency. MinLinA and MinLogA are always faster than LDG but, except for the *twitter* dataset, they are slower than the original ordering. As reported in Figure 4.3, we could not find any parameters with better results than local search, which is not ideal when the problem has local minima.

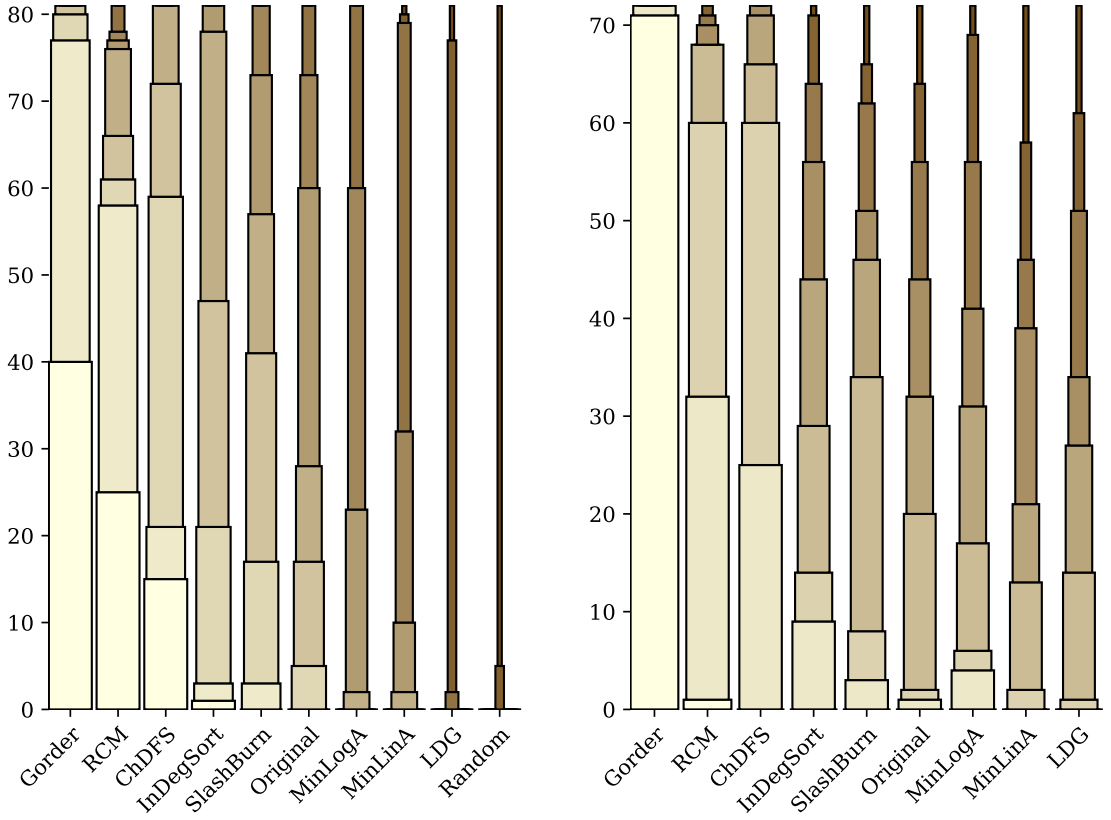
Degree-based orderings show medium performance

Both InDegSort and Slashburn use the degree of nodes as their main criterion. The experiments show that they outperform the default orderings, especially for larger datasets. For some algorithms such as BFS or NQ, they are less than 20% slower than Gorder. This indicates that cache misses are reduced when nodes of similar degree are copied together on a cache line. The original paper found similar results, though their implementation of Slashburn did not perform as well; the fact that we implemented a modified version (see Section 4.2.3) may be responsible for this discrepancy.

Some orderings outperform Gorder on specific algorithms

We also notice that some orderings perform particularly well on specific algorithms, and can even outperform Gorder. ChDFS ordering is the most efficient for DFS algorithm on all datasets. This is due to the close relation between these two processes: the algorithm explores the graph in the exact same way as the ordering is created. Likewise, RCM is a variation of a BFS that takes node degrees into account and it is the most efficient ordering for BFS algorithm.

Both also outperform Gorder for algorithms that are not as blatantly related: ChDFS is up to 10% more efficient for SCC on smaller datasets, and RCM is the most efficient for Diameter and SP. More generally, Figure 4.7 shows that these two orderings are among the three fastest ones in 75% of the experiments. The original paper has different results on that matter: RCM and ChDFS are the best alternatives as well, but they are always 10 to 20% slower than Gorder.



(a) Rankings in our experiments

(b) Rankings in the experiments of Wei et al.

Figure 4.7: **Rankings of ordering methods** in our experiments (left) and the experiments of Wei et al. (right). For each series of experiments, we rank the runtime performance of the orderings. This figure shows how many times an ordering ranks best (thickest, lightest bar), second-best, ..., to worst (thinnest, darkest bar). We have 81 experiments (9 algorithms times 9 datasets) and the original paper has 72, as it does not include *epinion*. Besides, we add Random ordering and remove Metis. Rankings of Wei et al. are inferred from their Figure 9, but speedups above 1.5 are not displayed so we consider them equal.

4.3.4 Comparison to the original paper

Our purpose here is to detect if there are significant discrepancies in performance between the original paper and our replication study. Figure 4.7 presents an aggregate view of the results grouped by ordering method. For each series of experiments (*i.e.* a given algorithm applied to a given dataset), we rank ordering methods from best to worst performance. The figures report how many times each ordering has been ranked in each position. For instance Gorder is ranked first in 40 of our 81 series of experiments.

Results for Gorder

In both the original study and our replication, Gorder ranks first overall. This shows that this ordering is the best choice in general. However, our study shows that Gorder is outperformed by different orderings in half of the experiments, while it is only outranked once in the original paper. Figure 4.7b thus leads us to think of Gorder as the perfect choice whereas Figure 4.7a establishes RCM and ChDFS as relevant challengers, with 24 and 16 first places respectively. This difference between the two papers is probably due to implementations: in our replication, ChDFS uses exactly the DFS algorithm. In particular, the nodes are visited in the same order, which leads to a quick execution of DFS. The original paper likely prevents this mechanism, for instance by shuffling the nodes at each step of the search, but no detail was given to hint at such a precaution.

Ranking of other orderings

The three best orderings are evidently the same in both studies, but there are some nuances for the other ones. Arguably, visualising these results does not always allow for exact ranking: in Figure 4.7b, InDegSort has more second and third places than Slashburn but fewer fourth and fifth places. There is no obvious way of deciding which is better, while Figure 4.7a clearly indicates InDegSort. The same issue happens between Original and MinLogA: we can say that their rank in the experiments of Wei et al. is equal.

The last nuance comes for the slowest orderings: in our study, LDG is only better than Random while MinLinA competes with MinLogA and Original. For Wei et al. on the other hand, LDG is better than MinLinA overall. This difference can be explained by the fact that Figure 9 of the original paper hides the speedup above 1.5, which makes the ranking unreliable for the poorest orderings.

In the end, both studies rank the orderings in a very similar way.

Limits of visualisation

The aggregate view of Figure 4.7 induces several approximations. First of all, there are 72 experiments in the original study and 81 experiments in ours, as we add the *epinion* dataset. This extra dataset is much smaller than the others and all its results in Figure 4.5 range in a 1.4 speedup, to be compared with more than 3 for the biggest datasets. Yet, the ranking of ordering methods on *epinion* is consistent with other datasets.

The original study does not test random orderings. This does not disturb the results as this method ranks last in most experiments. Similarly, our study omits Metis so we ignore it in Figure 4.7b as well. Moreover, the original paper hides precise information when a runtime exceeds 1.5 times the runtime of Gorder. We consider that all orderings above this bound rank equally.

The main issue of our visualisation is that it only shows the rank and hides the extent of runtime variations. This information is only visible in Figure 4.5, where a gap separates

two categories: faster orderings with Gorder, RCM, ChDFS, InDegSort and Slashburn, from slower orderings with the other ones. However, Figure 4.7 is useful to give an overall grade to each ordering method. If original ordering is taken as a limit between faster and slower orderings, we find the same gap again.

4.3.5 Cache miss

A cache-miss is a state of an execution when the data requested by the processor is not found in the cache memory. The program has to fetch the data in further cache levels or in main memory, which causes delays. Gorder capitalises on the intuition that if we cluster nodes that are frequently accessed together, higher levels of cache will hold more relevant data and thus make algorithms run faster.

To prove that the speedup associated to Gorder is due to cache-miss reduction, we compute the proportion of the total computation time spent in data retrieval. We use Unix *perftools* with the wrapper *ocperf*. It provides various hardware metrics such as the number of CPU cycles, branch predictions, cache misses...[3] Depending on the machine architecture, different metrics are available.

Table 4.3 shows the cache-miss rates at different levels, just like the Tables 3 and 4 of the original paper. The first column is the total number of L1-references which is the number of times a piece of data was required by the processor. A proportion of this data is not found (second column) and requested in intermediate levels of cache, until reaching L3 (third and fourth columns). The remaining data (last column) has to be retrieved in main memory. Note that each further level of cache roughly implies an extra factor 4 delay.

We observe that first-level cache references are similar for all orderings: the algorithms run in the exact same way regardless of the ordering of the nodes so they need to access the same amount of data (up to small fluctuations). However, the miss-rate in L1 reveals important variations: with Gorder, only 10% of the data is not directly available in L1, while it reaches 20% with Random or LDG orderings. The percentage of data requested in L3 is even more scattered, from 5% with Gorder to 20% with Random or LDG on *sdarc*. Finally, all the orderings have a low cache-miss rate (between 1.6 and 3.6%) on *flickr* and RCM has the smallest. The gap is more striking on *sdarc* where Random and LDG have 9% of cache-miss, three times as much as Gorder. This ratio is the proportion of data that had to be retrieved in main memory (RAM), which is about 60 times slower than the L1 cache[4].

In general, the ranking for cache-miss rates matches the ranking for runtime shown in Figure 4.7a. Gorder has the best results and RCM and ChDFS are close behind. MinLinA, MinLogA and Original orderings have high cache-miss rates for all levels. This shows that the speedup is indeed due to cache-miss reduction. When compared to Original ordering, Gorder reduces cache-miss rates to speed up the algorithms. Figure 4.2 shows that the total runtime is reduced by 15 to 50% on *sdarc*, but the CPU execution time is almost identical: it is the factor 3 reduction on cache stall that makes the algorithm faster.

[3] See https://perf.wiki.kernel.org. We select the following counters: the total time *task-clock*, *cpu-cycles*, *L1-dcache-loads* and *L1-load-misses* to measure the efficiency of the first layer of data cache (we are not interested in instruction cache here), *LLC-loads* and *LLC-load-misses* to measure the efficiency of the last cache layer, and metrics specifically designed to measure the impact of cache misses such as *cycles-l1d-pending* or *cycles-l3-miss* in the cycle-activity category.

[4] With a clock frequency of 4GHz, a cycle takes $c = 1/(4 \cdot 10^9)s = 0.25ns$, so latency is $4c = 1ns$ for L1 and $42c + 51ns \simeq 62ns$ for RAM, with the formulas given at https://www.7-cpu.com/cpu/Skylake.html.

Order	L1-ref (10^9)	L1-mr	L3-ref (10^9)	L3-r	Cache-mr
Original	29	15.9 %	2.8	9.8 %	2.5 %
Random	30	20.2 %	4.1	13.6 %	3.6 %
MinLA	29	16.2 %	3.3	11.4 %	2.5 %
MinLogA	28	15.9 %	3.1	10.7 %	2.5 %
RCM	30	11.5 %	1.8	5.9 %	**1.6 %**
InDegSort	28	14.7 %	2.5	9.1 %	2.2 %
ChDFS	29	12.8 %	2.1	7.2 %	1.8 %
SlashBurn	28	14.8 %	2.6	9.3 %	2.2 %
LDG	30	19.2 %	3.7	12.4 %	3.2 %
Gorder	28	**10.3 %**	**1.4**	**5.0 %**	1.7 %

(a) On *flickr* dataset.

Order	L1-ref (10^9)	L1-mr	L3-ref (10^9)	L3-r	Cache-mr
Original	1885	19.0 %	303	16.0 %	6.8 %
Random	1886	23.4 %	397	21.0 %	9.0 %
MinLA	1893	21.2 %	341	18.0 %	7.1 %
MinLogA	1885	20.7 %	330	17.5 %	6.9 %
RCM	1885	11.0 %	139	7.4 %	3.7 %
InDegSort	1779	15.0 %	198	11.1 %	6.0 %
ChDFS	1863	11.8 %	153	8.2 %	4.3 %
SlashBurn	1784	15.3 %	203	11.4 %	6.0 %
LDG	1886	22.9 %	387	20.5 %	8.8 %
Gorder	1816	**9.3** %	**104**	**5.7 %**	**3.1 %**

(b) On *sdarc* dataset.

Table 4.3: **Cache statistics measured for Pagerank algorithm.** L1-ref (references): number of times a piece of data was required by the processor and searched in level 1 of cache. L1-mr (miss-rate): proportion of data that was not found in L1. L3-ref: number of references to the third (lowest) level of cache after data was not found in levels 1 and 2. L3-r (ratio): proportion of data that was not found in L1 nor L2, then searched in L3. Cache-mr: proportion of data that was not found in cache (L1, L2 or L3) and had to be retrieved in main memory. Compare to Tables 3 and 4 in Wei et al. [2016].

4.4 Discussion

Our experiments replicate the ones proposed by Wei et al. [2016] but some aspects are not discussed in sufficient detail in the original paper to allow for immediate replication. For instance, the algorithms NQ and DS only have a succinct description, which may explain why their performances reported in our Figure 4.2 do not align perfectly with Figure 1 of the original paper. Similarly, we were not able to tune the simulated annealing procedure correctly, which questions the relevance of our experiments with MinLinA and MinLogA. Nonetheless, most of these technical issues have been solved or circumvented thanks to the answers of Hao Wei to our questions.

Above all, this study replicates the main observation of the original paper: Gorder reduces cache latency significantly and is the best performer among all the ordering methods under study, as shown in Figure 4.7. Its consistent efficiency on all algorithms and datasets suggests that it could speed up other graph algorithms as well.

The only important difference with the original paper is that RCM and ChDFS follow

closely behind Gorder and even outperform it in half of our experiments. These orderings are based on graph searches that can be computed with simple and scalable algorithms, leading to the very quick execution times reported in Table 4.2. On the other hand, computing Gorder requires a complex procedure and a lot of time. It has been pointed out by Balaji and Lucia [2018] that this high overhead can only be amortised if algorithms are run thousands of times. In the case where networks evolve and require constant recomputation of the node ordering, Gorder needs to be adapted to integrate the modifications without running the whole process again. A parallel version of Gorder could reduce this problem.

Beyond algorithm speed-up, the contribution of Wei et al. is an efficient framework that could be used for other purposes. For example, Chapter 3 shows that graph compression also benefits from node orderings that cluster nodes with high proximity; Gorder could be an input for such existing methods. It would also be interesting to investigate how different types of real-world datasets, as described by Milo et al. [2004], behave when a new ordering is applied.

4.5 Conclusion

Our replication of the paper of Wei et al. [2016] shows that Gorder is an efficient cache optimisation for various standard algorithms. We confirm its superiority for networks ranging from 30 million to 2 billion edges, and the hardware measurement tools prove that this is due to reduced cache stall. However, orderings such as DFS are among the best performers in spite of their much simpler design. The fact that the computation of Gorder does not scale linearly makes it impractical for larger graphs. Hence, there is a trade-off between the overhead time required to compute Gorder and the substantial speedup that it provides for subsequent graph algorithms.

Chapter 5

Orderings for reduced operations

Listing the triangles of an undirected graph

Summary

One crucial problem in graph mining is to identify specific structures within the network. As explained in the introduction, the datasets that we use describe a network by its nodes and its edges, which consist of two linked nodes. The occurrence of three linked nodes, also called a triangle, is an important feature for data science tasks in various scientific domains that use network data. Listing triangles is therefore a key task that has been addressed widely in the literature. In the context of large real-world graphs, the current fastest algorithms use node orderings. First, the nodes are given indices in a specific order. Then, the edges are directed from lower to higher node indices: they all point in one specific direction. Instead of scanning edges until three of them form a triangle, the algorithm follows the direction of the edges, and triangles are identified faster. To simplify, the algorithm explores paths of length two: from an origin node, through a middle node, reaching a destination node; if the origin and the destination are connected by a third edge, a triangle has been found. The execution time of the algorithm depends on the number of paths of length two; but recall that the paths are defined by the direction of edges, which are given by the indices of the nodes. The choice of ordering will thus impact the time that the algorithm takes to find all the triangles.

While state-of-the-art triangle listing algorithms use this technique to accelerate their execution, they use basic orderings that provide general mathematical guarantees. On the contrary, our work studies the precise correlation between the choice of ordering and the execution time of the algorithm. We define new orderings that exploit this correlation: even though we prove that finding optimal orderings is theoretically difficult, we propose a trade-off between the time spent to compute the ordering and the time saved while listing the triangles. Our experiments on large real-world networks confirm that the new orderings significantly accelerate the listing of triangles.

Contributions

* Introduce cost functions that relate the node orderings to the running time of triangle listing algorithms.
* Prove that finding an optimal ordering that minimises either of these costs is computationally difficult (NP-hard).
* Expose a gap in the combinations of algorithm and ordering considered in the literature, and bridge it with three heuristics producing orderings with low costs.
* Show experimentally that the resulting combinations of algorithm and ordering outperform state-of-the-art running times for the triangle listing task.
* Release an efficient open-source implementation of all considered methods.
  https://github.com/lecfab/volt

Publications & talks

* *Tailored vertex ordering for faster triangle listing in large graphs*
  Lécuyer, Jachiet, Magnien, and Tabourier [2023a], ALENEX.
* Presented at FRCCS'22, MLG'22, ALENEX'23.
* Seminars in Milan and Vancouver.

5.1 Introduction

5.1.1 Motivation

Small connected subgraphs are key to identifying families of real-world networks, and they are interchangeably called patterns, motifs or graphlets depending on the field of research. Indeed, they are used for descriptive or predictive purposes across various fields such as biology, linguistics, engineering or sociology, and a large amount of work addresses their efficient mining.

For Milo et al. [2002], knowing which patterns appear the most frequently gives away the category to which a real-world network belongs: in their experiments, the frequent patterns within information processing networks are the same even though they represent distinct situations such as protein regulation or electronic circuits, and they differ from frequent patterns of energy processing networks such as food chains. Sporns and Kötter [2004] suggest that the brain functionalities derive from the variety of subgraphs (a subset of nodes with *some* of the edges that they share) within a limited number of structural patterns, that they define as induced subgraphs (a subset of nodes with *all* the edges that they share). Pržulj [2007] measures the similarity between networks by counting their graphlets, and can then assess which synthetic graph model is more suitable for which type of real-world network. Analysing the motifs of a network built upon the co-occurrence of words in a sentence, Biemann et al. [2016] are able to infer the grammatical properties of the words, such as their part of speech or their syntactic function. Looking at evolving networks, Valverde and Solé [2005] suggest that software architectures emerge from a sequence of duplication and rewiring that create the same motifs as in some biological networks.

In sociology, characterising networks with specific structural patterns has been a focus of interest for a long time, as it is even present in the works of early 20^{th} century sociologists such as Simmel [1908] on structural analysis [Wellman, 1988]. Consequently, it is a common practice in social network analysis to describe interactions between individuals using local patterns [Holland and Leinhardt, 1976, Wasserman et al., 1994]. The recent ability to count and list small patterns efficiently allows for the characterisation of various

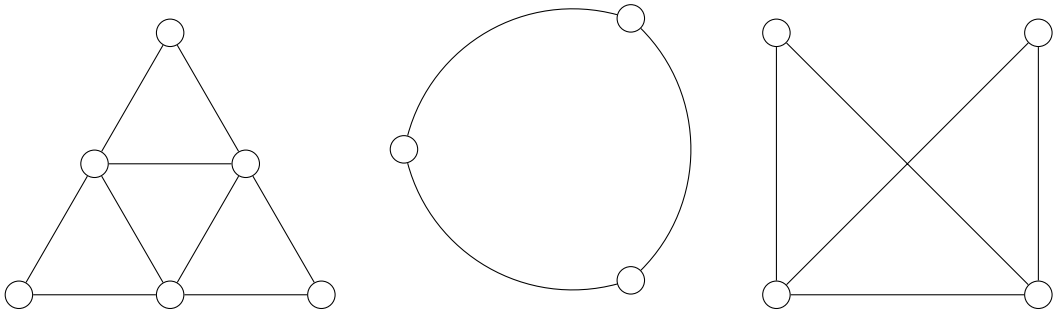


Figure 5.1: **How many triangles are there?** Answer: 4, 1 and 2. In our work, a triangle is a set of three connected nodes. On the left, the outer triangle does not count because its nodes are not connected; it looks like a triangle because the drawings of unrelated edges are parallel. In the middle, it is a triangle even though it looks circular. On the right, the 3 small triangles do not count because one of their corners is not a node of the graph; they look like triangles because the drawing of other edges cross.

types of social networks on a large scale [Choobdar et al., 2012, Charbey and Prieur, 2019]. In particular, listing elementary patterns such as triangles and 3-motifs is a stepping stone in the analysis of the structure of networks and their dynamics [Faust, 2010]. For instance, the closure of a triplet of nodes to form a triangle is thought to be a driving force for the evolution of social networks [Leskovec et al., 2008, Sintos and Tsaparas, 2014].

In spite of these numerous applications for the analysis of real-world networks, there is one topic that is sometimes mistakenly associated with triangle listing: surface triangulation, which consists in decomposing a two-dimensional surface in triangles. This process, used for instance in 3D rendering, creates a graph full of triangles that we could then count and list. However, this graph is more specific than the ones that we study in the sense that its nodes are embedded in a metric space. Our nodes do not have coordinates, so the triangle is an abstract triplet of neighbours as opposed to a geometric triangle that can be drawn. The number of triangles is not influenced by parallel, curved or crossing edges, which means that the three graphs of Figure 5.1 contain four, one and two triangles respectively.

5.1.2 Related works

In terms of algorithm, the task of listing triangles is rather simple: one needs to take all sets of three nodes and check whether they are connected. Even supposing that checking the adjacency of two nodes can be done in constant time, this strategy would require $\Theta(n^3)$ operations and lead to an unacceptable execution time for large networks, as explained in Chapter 1. Over around fifty years of research in this field of graph mining, more sophisticated algorithms have been discovered, and they specialised into different ways of finding triangles: some algorithms are designed to count the triangles, while other list them; some do it with knowledge of the whole graph while other analyse subgraphs turn by turn. Al Hasan and Dave [2018] provide a review of the different tasks of triangle mining and the influential algorithms that exist for each of them.

When asking to find triangles, the first precision to give is whether the goal is to list them or to count them. Our work focuses on the former, where the expected output is a list of triplets. Yet, several methods have been designed for the latter, and Hu et al. [2021] show that they can also benefit from node orderings. Counting triangles is an easier problem than listing: having the list of triangles is sufficient to know how many there are, but the reverse is not true. In particular, counting benefits from the matrix representation of graphs as described by Itai and Rodeh [1977]. Consider the adjacency matrix A where $A_{uv} = 1$ if there is an edge between nodes u and v, 0 otherwise. Multiplying A by itself gives $B = A^2$ and the value B_{uv} counts the distinct paths of length two between u and v. Two nodes share a triangle when they have both a path of length one and a path of length two between them. Thus, once B is computed, the total number T of triangles is the number of paths of length two (B_{uv}) between nodes that are connected (A_{uv}):

$$T = \sum_{u,v \in V} A_{uv} \cdot B_{uv}$$

The sum is computed in time $\Theta(n^2)$ or even $\Theta(m)$ depending on the data-structure, but computing B is more costly: the complexity of matrix multiplication is noted $\mathcal{O}(n^\omega)$, where $2 \leq \omega \leq 3$. The best known theoretical algorithms have exponent under 2.4, but practical implementations use exponents over 2.8, which makes the matrix multiplication method close to the $\Theta(n^3)$ naive algorithm. However, matrix multiplication is the speciality of Graphics Processing Units (GPU). These processors are designed to apply

simple instructions over thousands of small pieces of data in parallel, as opposed to Central Processing Units (CPU) that can apply complex instructions over a single large piece of data [see TutorialsPoint]. Using GPUs, big matrices can be multiplied by handling submatrices in parallel, which speeds up the execution. This is one of the methods leveraged by Wang et al. [2016] to compare competitive triangle counting algorithms on GPUs.

Another approach to triangle mining consists in prioritising speed over accuracy: instead of reading the whole graph and determining the exact number of triangles, a sampling procedure gives an estimate of this number without looking at all the nodes. The approach of Seshadhri et al. [2014] consists in sampling wedges (paths of length two) and checking whether they close into a triangle. Checking all the wedges gives a precise triangle count, but checking only a small portion of them can provide a high accuracy in much faster execution time. Turk and Turkoglu [2019] further reduce the sampling space by directing the edges and only considering a wedge if its middle node is a predecessor of the other two nodes. Edge directions are chosen to minimise the number of such directed wedges, which is similar to optimising the function C^{++} described in Section 5.4. Sampling procedures are backed by theoretical results in random walk theory [Bressan, 2021] and sampling complexity [de Lima et al., 2022] that guarantee a certain accuracy for a given sample size.

Finally, the literature distinguishes triangle listing methods depending on their memory requirements. In our work, we consider large graphs that still fit in the main memory (see the description of our hardware in introduction). On the other hand, specific methods exist for graphs that do not fit in main memory: they partition the graph into subgraphs that are loaded and processed in turns: Chu and Cheng [2011] focus on reducing the costly I/O operations (reading and writing in hard memory), while Arifuzzaman et al. [2019] address time and space efficiency. Alternatively, streaming approaches consider that the edges can only be read sequentially, which requires to store previous information in hash tables [Becchetti et al., 2008, Gou and Zou, 2021] or in more cache-friendly structures [Tangwongsan et al., 2013].

As for in-memory exact triangle listing, the first effort to design efficient algorithms appeared in Chiba and Nishizeki [1985]. Based on the observation that real-world graphs generally have a heterogeneous degree distribution, later contributions by Schank and Wagner [2005] and Latapy [2008] showed how ordering vertices by degree or core value accelerates the listing. Such orderings create an orientation of edges so that nodes that are costly to process are not processed many times. Unifying descriptions of this method have been proposed alongside comparison of orderings, by Xiao et al. [2017] for asymptotic random graphs and by Ortmann and Brandes [2013] for real-world graphs. Successful extensions to cliques of arbitrary size were proposed by Uno [2012], Danisch et al. [2018], Li et al. [2020]. However, only degree and core orderings have been exploited, although their properties are not specifically tailored for the triangle listing problem. Since other types of orderings benefit a variety of graph problems, as surveyed in Chapter 3, the main purpose of this chapter is to find a general method to design efficient vertex orderings for triangle listing.

5.1.3 Contributions and outline

In this chapter, we show how vertex ordering directly impacts the running time of the two fastest existing in-memory triangle listing algorithms. First, we introduce cost functions that relate the vertex ordering to the running time of each algorithm. We prove that finding an optimal ordering that minimises either of these costs is NP-hard. Then, we expose a gap in the combinations of algorithm and ordering considered in the literature,

and we bridge it with three heuristics producing orderings with low corresponding costs. Our heuristics reach a compromise between their running time and the quality of the ordering obtained, in order to address two distinct tasks: listing triangles with or without taking into account the ordering time. Finally, we show that our resulting combinations of algorithm and ordering outperform state-of-the-art running times for either task. We release an efficient open-source implementation [1] of all considered methods.

Section 5.2 presents state-of-the-art methods to list triangles. In Section 5.3, we prove the NP-hardness of two minimisation problems over the set of orderings and discuss relaxations and approximations. In Section 5.4, we analyse the cost induced by a given ordering on these algorithms and propose several heuristics. The experiments of Section 5.5 show that our methods are efficient in practice and improve the state of the art.

5.2 State of the art

5.2.1 Triangle listing algorithms

Targeting nodes or edges

Ortmann and Brandes [2013] have identified two families of triangle listing algorithms that rely on two basic graph operations: *adjacency testing*, and *neighbourhood intersection*.

The first family considers a triangle as a node with adjacent neighbours. Algorithms tree-lister [Itai and Rodeh, 1977], node-iterator [Schank and Wagner, 2005] and forward [Schank and Wagner, 2005] belong to this category. They sequentially consider each vertex u as a seed, and process all pairs $\{v, w\}$ of its neighbours; if they are themselves adjacent, $\{u, v, w\}$ is a triangle. The efficiency of this method depends on the complexity of testing the adjacency of two nodes (v and w). More details on the different complexities involved by a selection of data-structures are provided below.

In contrast, methods of the neighbourhood intersection consider a triangle as an edge whose extremities share a neighbour. Algorithms edge-iterator [Schank and Wagner, 2005], compact-forward [Latapy, 2008] and K3 [Chiba and Nishizeki, 1985] belong to this category, as well as some algorithms that list larger cliques [Makino and Uno, 2004, Danisch et al., 2018, Li et al., 2020]. They sequentially consider each edge (u, v) as a seed; each common neighbour w of u and v forms a triangle $\{u, v, w\}$. Here, the efficiency depends on the complexity of computing the intersection of two lists of different size. The impact of various data-structures is also presented below.

Node orderings avoid redundancy

In naive versions of both adjacency testing and neighbourhood intersection, finding a triangle (u, v, w) does not prevent from finding triangle (v, w, u) at a later step. Several techniques exist to avoid this unwanted redundancy. The K3 algorithm introduced by Chiba and Nishizeki [1985] removes a node from the graph once all the triangles comprising it have been found; this prevents duplicates but requires a data-structure with linked lists to update the graph, leading to extra space and time consumption. The forward algorithm by Schank and Wagner [2005] uses extra storage for each node to remember which of its neighbours have already been treated, thus ensuring that only one permutation of each triangle is found.

[1] https://github.com/lecfab/volt

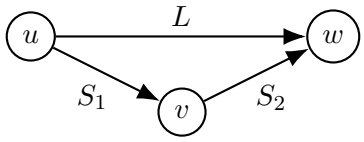


Figure 5.2: **Directed triangle** with the unified notations proposed by Ortmann and Brandes [2013]. The edges are directed according to an ordering π such that $\pi_u < \pi_v < \pi_w$.

Recent papers use a node ordering, explicitly or not, which is best described using the framework developed by Ortmann and Brandes [2013]: a total ordering π is defined over the vertices, and the triple (u, v, w) is only considered a valid triangle if $\pi_u < \pi_v < \pi_w$. This guarantees that each triangle is listed only once: as illustrated in Figure 5.2, vertices in any triangle of the DAG G_π appear in one and only one of 3 positions: u is first, v is second, w is third; the same holds for edges: L is the long edge, and S_1 and S_2 are the first and second short edges. This leads to three variants of adjacency testing (seed vertex v or w instead of u) and of neighbourhood intersection (seed edge L or S_2 instead of S_1).

Data-structures influence complexity

The above classification reduces triangle listing to two algorithmic problems: adjacency testing requires to check the *existence* of an element x in a list L, and neighbourhood intersection requires to compute the *intersection* between two lists L_1 and L_2. The existence and intersection problems have different complexities depending on their implementation. The literature on triangles mainly focuses on two methods: standard search and hash tables.

A *standard search* consists in consecutively checking all the elements of a list L. For the existence problem, it takes $\mathcal{O}(|L|)$ operations. For the intersection problem, each element of L_1 needs to be searched through L_2, which corresponds to a complexity $\mathcal{O}(|L_1| \cdot |L_2|)$. If the lists are sorted, it reduces to $\mathcal{O}(|L_1| + |L_2|)$ because an element of L_1 cannot appear in L_2 after larger elements. This method is used in compact-forward [Latapy, 2008] with seed edge S_2, leading to the following complexity for triangle listing:

$$\mathcal{O}\Big(m + \sum_{v \in V} d_v \cdot d_v^-\Big)$$

This quantity is known to be minimised by the degree ordering, as Arifuzzaman et al. [2019] demonstrate in their Theorem 4.3. Note that a binary search, which also requires the lists to be sorted, would lead to an existence checking in $\log_2 |L|$ steps and an intersection in $\mathcal{O}(|L_1| \cdot \log |L_2|)$ assuming $|L_1| \leq |L_2|$, but the sorted standard search is preferred in the literature.

A *hash table* allows to check in constant time whether x belongs to L, or to compute an intersection in time $\mathcal{O}(\min(|L_1|, |L_2|))$ by going through the values of the shorter list and checking (in constant time) if they also belong to the longer list. Storing all the edges of a graph in a hash table requires $\Theta(m)$ space but allows to find the triangles (for seed edge L) in:

$$\mathcal{O}\Big(m + \sum_{(u,v) \in E} \min(d_u^+, d_v^-)\Big)$$

Considering complexity only, hash tables seem to be the most efficient option, and Schank and Wagner [2005] indeed show that the edge-iterator-hashed and forward-hashed

methods achieve the smallest number of algorithmic operations. However, it is established that such implementations are the least efficient in practice, as Schank and Wagner [2005] point out: *"they require advanced data structures which experimentally result in a high constant overhead"*. In our experiments on small datasets, we confirmed that this method was up to 50 times slower than the sorted standard search. The randomised location of data in hash tables makes the cache system inefficient, which may explain the slow access (see Section 2.3 and Chapter 4).

Besides, standard search can be made very efficient for neighbourhood intersection with the use of a *boolean table*. We present this strategy in Algorithms 2 and 3 with the notations of Figure 5.2 for the vertices. They initialise the boolean array B to false (line 1), consider a first vertex (line 2) and store its neighbours in B (line 3); then, for each of its neighbours (line 4), they check if their neighbours (line 5) are in B (line 6), in which case the three vertices form a triangle (line 7). B is reset (line 8) before continuing with the next vertex.

Algorithm 2 – A++ (or L+n)

1: **for** each vertex v **do** $B[v] \leftarrow$ False
2: **for** each vertex w **do**
3: **for** $v \in N_w^-$ **do** $B[v] \leftarrow$ True
4: **for** $u \in N_w^-$ **do**
5: **for** $v \in N_u^+$ **do**
6: **if** $B[v]$ **then**
7: **output** triangle$\{u, v, w\}$
8: **for** $v \in N_w^-$ **do** $B[v] \leftarrow$ False

Complexity:

$$\Theta\Big(m + \sum_{(u,w)\in E_\pi} d_u^+\Big) = \Theta\Big(\sum_{u\in V} {d_u^+}^2\Big)$$

Algorithm 3 – A+− (or S_1+n)

1: **for** each vertex w **do** $B[w] \leftarrow$ False
2: **for** each vertex u **do**
3: **for** $w \in N_u^+$ **do** $B[w] \leftarrow$ True
4: **for** $v \in N_u^+$ **do**
5: **for** $w \in N_v^+$ **do**
6: **if** $B[w]$ **then**
7: **output** triangle$\{u, v, w\}$
8: **for** $w \in N_u^+$ **do** $B[w] \leftarrow$ False

Complexity:

$$\Theta\Big(m + \sum_{(u,v)\in E_\pi} d_v^+\Big) = \Theta\Big(m + \sum_{v\in V} d_v^+ d_v^-\Big)$$

Boolean tables can provide constant-time adjacency testing when triangle listing algorithms search through the same list several times. If we look at Algorithm 3, the neighbourhood N_u^+ of a node u is consecutively intersected with the neighbourhoods N_v^+ of different nodes v. Registering the elements of N_u^+ in the boolean array allows for checking in constant time if a neighbour w of v is also a neighbour of u. Maintaining this register amounts for $\Theta(d_u^+)$ for each node u, which is $\Theta(m)$ in total. The K3-algorithm of Chiba and Nishizeki [1985] uses this method, which is called +n by Ortmann and Brandes because storing the boolean array requires an extra $\Theta(n)$ space. The fastest methods reported by Ortmann and Brandes [2013] and Danisch et al. [2018] use a boolean table.

In the rest of this chapter, we therefore only consider triangle listing algorithms that use neighbourhood intersection and a boolean array. Algorithm 2 corresponds to L+n of Ortmann and Brandes; we call it A++ because of the two "+" (referring to out-degrees) involved in its complexity. Algorithm 3 corresponds to S_1+n of Ortmann and Brandes; we call it A+−[2]. Their complexities are given in Property 1; since they depend on the in- and out-degree of vertices, the choice of ordering will impact the running time of the algorithms.

[2] A third natural variant exists: A−− or S_2+n. We ignore it here since its complexity is equivalent to the one of A++ upon reversing ordering.

Property 1 (Complexity of A++ and A+−) *The time complexity of* A++ *is* $\Theta(\sum_{u\in V} {d_u^+}^2)$. *The time complexity of* A+− *is* $\Theta\left(m + \sum_{v\in V} d_v^+ d_v^-\right)$.

Proof: In both algorithms, the boolean table B requires n initial values, m set and m reset operations, which is $\Theta(m)$ assuming that $n \in \mathcal{O}(m)$. In A++, a given vertex u appears in the loop of line 4 as many times as it has a successor w; every time, a loop over each of its successors v is performed. In total, u is involved in $\Theta({d_u^+}^2)$ operations. Similarly, in A+−, a given vertex v appears in the loop of line 4 as many times as it has a predecessor u; every time, a loop over each of its successors w is performed. In total, v is involved in $\Theta(d_v^+ d_v^-)$ operations. Using the boolean table B, line 6 of both algorithms takes constant time. The term m is omitted in the complexity of A++ as $\sum_{u\in V} {d_u^+}^2 \geq \sum_{u\in V} d_u^+ = m$, but not in A+− as $\sum_{v\in V} d_v^+ d_v^-$ can be lower than m. □

5.2.2 Node orderings and complexity bounds

Ortmann and Brandes [2013] order the vertices by non-decreasing degree or core value. In their experimental comparison, they test several algorithms as well as A++ and A+−, each with degree ordering, core ordering, and with the original ordering of the dataset. They conclude that the fastest method is A++ with core or degree ordering: core is faster to list triangles when the ordering is given as an input, and degree is faster when the time to compute the ordering is also taken into account.

Danisch et al. [2018] also use core ordering in the more general problem of listing k-cliques. For triangles ($k = 3$), their algorithm is equivalent to A+−, and they show that using core ordering outperforms the methods of Chiba and Nishizeki [1985], Makino and Uno [2004] and Latapy [2008].

With these two orderings, it is possible to obtain upper-bounds for the time complexity in terms of graph properties. Chiba and Nishizeki [1985] show that K3 with degree ordering has a complexity in $\mathcal{O}(m \cdot \alpha(G))$, where $\alpha(G)$ is the arboricity of graph G. With core ordering, node-iterator-core [Schank and Wagner, 2005] and kClist [Danisch et al., 2018] have complexity $\mathcal{O}(m \cdot c(G))$, where $c(G)$ is the core value of graph G. These bounds are considered equal by Ortmann and Brandes [2013], following the proof by Zhou and Nishizeki [1994] that $\alpha(G) \leq c(G) \leq 2\alpha(G) - 1$. However, we focus in this chapter on the complexities expressed in Algorithms A++ and A+− as we will see that they describe the running time more accurately.

5.3 Hardness of cost minimisation

In this section, we study the hardness of different problems related to triangle listing. For this purpose, we introduce the following costs that appear in the complexity formulas of Algorithms 2 and 3. Recall that the initial graph is undirected and that the orientation of the edges is given by the ordering π, which partitions neighbours into successors and predecessors.

Definition 2 (Cost induced by an ordering) *Given an undirected graph G, the costs C^{++} and C^{+-} induced by a vertex ordering π are defined by:*

$$C^{++}(\pi) = \sum_{u\in V} d_u^+ d_u^+ \qquad C^{+-}(\pi) = \sum_{u\in V} d_u^+ d_u^-$$

In order to speed up the execution of these algorithms, we wish to obtain orderings that minimise these quantities, which are the term of their complexity that depends on orderings. However, we will see that obtaining exact minimum solutions is hard, and we will discuss approximability and relaxations of the problem from orderings to orientations.

5.3.1 NP-hardness of the C^{+-} problem

To prove that it is NP-hard to find an ordering that minimises C^{+-}, we will use the equivalent notation of a total order $\prec$ instead of the permutation π: given a graph G and an order $\prec$ on the vertices of G, we define $succ_\prec(u)$ (respectively $pred_\prec(u)$) as the set of neighbours v of u such that $u \prec v$ (resp. $v \prec u$). For any subset of vertices W, we note $C^{+-}_\prec(W) = \sum_{u \in W} |succ_\prec(u)| \cdot |pred_\prec(u)|$. Using this definition we formalise the following problem:

Problem 1 (C^{+-}) *Given an undirected graph $G = (V, E)$ and an integer K, is there an order $\prec$ on the vertices such that $C^{+-}_\prec(V) \leq K$?*

Let us introduce the NAE3SAT+ problem, known to be NP-complete by Schaefer [1978]'s dichotomy theorem. We will show that this problem can be reduced to the C^{+-} problem, thus proving that C^{+-} is NP-hard. Note that a sketch of proof was given online by Rudoy [2017] but, as far as we know, it has never been published. This section gives a new simpler proof of Theorem 1 below.

Problem 2 (**NAE3SAT+**) *Not-All-Equal Positive Three-Satisfiability. Given a formula $\phi = c_1 \wedge \cdots \wedge c_m$ in conjunctive normal form where each clause consists in three positive literals, is there an assignment to the variables satisfying ϕ such that in no clause all three literals have the same truth value?*

Theorem 1 *C^{+-} is NP-hard.*

Definition 3 *Let ϕ be an instance of NAE3SAT+ with variables $x_1, \ldots, x_n$ and clauses $c_1, \ldots, c_m$, where clause c_j is of the form $l^1_j \vee l^2_j \vee l^3_j$. We define a graph G_ϕ by creating three connected vertices L^1_j, L^2_j, L^3_j representing the literals of each clause c_j; additionally, a vertex X_i is created for each variable x_i and connected to all the L^a_j such that $l^a_j = x_i$. More formally, $G_\phi = (V_\phi, E_\phi)$ with:*

* $V_\phi = \{X_i \mid i \in [\![1, n]\!]\} \cup \{L^1_j, L^2_j, L^3_j \mid j \in [\![1, m]\!]\}$
* $E_\phi = \left\{\{L^1_j, L^2_j\}, \{L^1_j, L^3_j\}, \{L^2_j, L^3_j\} \mid j \in [\![1, m]\!]\right\} \cup \left\{\{X_i, L^a_j\} \mid x_i = l^a_j\right\}$

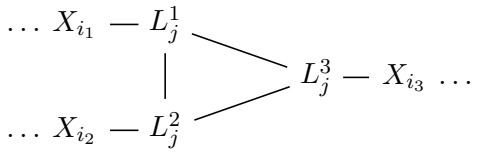


Figure 5.3: **Gadget included in G_ϕ for each clause** c_j of an instance ϕ of NAE3SAT+.

Proposition 1 ($\Longrightarrow$) *Given an instance ϕ of NAE3SAT+ with m clauses and the associated graph G_ϕ, if ϕ is satisfiable then there exists an order $\prec$ on V_ϕ such that $C^{+-}_\prec(V_\phi) \leq 2m$.*

Proof: Let ϕ be a satisfiable instance of NAE3SAT+ with the above notations. Take a valid assignment and let us note k the number of variables set to true. There exist indices $i_1, \dots, i_n$ such that $x_{i_1}, \dots, x_{i_k} = true$ and $x_{i_{k+1}}, \dots, x_{i_n} = false$, and for each clause c_j, there are indices $t_j, a_j, f_j \in \{1, 2, 3\}$ such that $l_j^{t_j} = true$, $l_j^{f_j} = false$ and $l_j^{a_j}$ has any value. Now construct the following order on V_ϕ, so that true variables come first, then in each clause the false literal comes before the true one, and the false variables are at the end:

$$
\begin{aligned}
& X_1 \prec \cdots \prec X_k && \text{True variables} \\
\prec\ & L_1^{f_1} \prec \cdots \prec L_m^{f_m} && \text{False literals} \\
\prec\ & L_1^{a_1} \prec \cdots \prec L_m^{a_m} && \text{Other literals} \\
\prec\ & L_1^{t_1} \prec \cdots \prec L_m^{t_m} && \text{True literals} \\
\prec\ & X_{k+1} \prec \cdots \prec X_n && \text{False variables}
\end{aligned}
$$

If a given variable x_i is true, the associated vertex X_i has only successors, if it is false it has only predecessors, so in both cases $C_\prec^{+-}(\{X_i\}) = 0$. For a given clause c_j, the variable $l_j^{f_j}$ is false so the corresponding X_i is a successor of $L_j^{f_j}$, which also has successors $L_j^{a_j}$ and $L_j^{t_j}$, but no predecessor. Similarly, $L_j^{t_j}$ has no successor; thus $C_\prec^{+-}(\{L_j^{f_j}, L_j^{t_j}\}) = 0$. Now $L_j^{a_j}$ has one predecessor $L_j^{f_j}$, one successor $L_j^{t_j}$, and one neighbour X_i that is a predecessor if x_i is true, otherwise a successor; in both cases, $C_\prec^{+-}(\{L_j^{a_j}\}) = 2$. The only vertices with a non-negative cost are the $L_j^{a_j}$, so the sum over all m clauses gives $C_\prec^{+-}(V_\phi) = 2m$. □

Proposition 2 ($\Longleftarrow$) *Given an instance ϕ of NAE3SAT+ with m clauses and the associated graph G_ϕ, if there exists an order $\prec$ on V_ϕ such that $C_\prec^{+-}(V_\phi) \leq 2m$ then ϕ is satisfiable.*

Proof: Conversely, consider an order $\prec$ on V_ϕ such that $C_\prec^{+-}(V_\phi) \leq 2m$. For all j, define $f_j, a_j, t_j \in \{1, 2, 3\}$ such that $L_j^{f_j} \prec L_j^{a_j} \prec L_j^{t_j}$; then $L_j^{a_j}$ has one successor, one predecessor, and one other neighbour X_i, so its cost is 2. As G_ϕ contains m such independent triangles, $C_\prec^{+-}(\{L_1^{a_1}, \dots, L_m^{a_m}\}) = 2m$. To ensure $C_\prec^{+-}(V_\phi) \leq 2m$, all the other vertices must have either only predecessors or only successors. If vertex X_i has successors only, assign x_i to true; if X_i has predecessors only, assign x_i to false. For all j, $L_j^{f_j}$ has at least 2 successors ($L_j^{a_j}$ and $L_j^{t_j}$) so its corresponding X_i has to be a successor, which means $x_i = l_j^{f_j}$ is false; similarly, $l_j^{t_j}$ is true. Each clause thus has one true and one false literal, so ϕ is satisfied. □

5.3.2 NP-hardness of the C^{++} problem

To prove the NP-hardness of the C^{++} problem, we will introduce a generalisation of this problem to weighted vertices. Then, we will find criteria for optimum orderings for this weighted problem. We will show that that the weighted problem is reducible to C^{++}, and finally that the NP-hard problem of Set Cover is reducible to the weighted problem.

The C^{++} problem consists in finding a permutation π that minimises the induced cost. Equivalently, we will be looking for a total order $\prec$ over the vertices that minimises the cost function of interest. Intuitively, we can think of $\prec$ as the order in which we eliminate vertices: each time we eliminate a vertex with an out-degree d we pay a cost of d^2 and the cost of an order is the cost of eliminating all vertices. For each vertex u, we will note $succ_\prec(u)$ its number of successors, which is the number of its neighbours appearing after u in the order $\prec$. When u is eliminated, we pay a cost $|succ_\prec(u)|^2$, which leads to the following reformulation of the C^{++} problem:

Problem 3 (C^{++}) *For a given undirected graph $G = (V, E)$ and an integer K, does there exist an order $\prec$ of the vertices such that $\sum_{u \in V} |succ_{\prec}(u)|^2 \leq K$?*

Theorem 2 *C^{++} is NP-hard.*

Generalisation of C^{++} to weighted vertices

As an intermediate step for the reduction between the Set Cover problem and C^{++}, we will use a generalisation of C^{++} to weighted vertices: this is the weighted-C^{++}, presented below:

Problem 4 (**weighted-C^{++}**) *Given an undirected graph $G = (V, E)$, a vertex-weighting function $w : V \to \mathbb{N}$ and an integer K, does there exist an order $\prec$ of the vertices such that $\sum_{u \in V} (|succ_{\prec}(u)| + w(u))^2 \leq K$?*

Given a graph G with the vertex weighting function w and an order $\prec$, the *cost* is the function $\sum_{u \in V} (|succ_{\prec}(u)| + w(u))^2$ applied to the graph with that order. The *optimal cost* of a graph is the minimal cost achievable by any order. Notice that the C^{++} problem is a special case of the weighted-C^{++} problem where weights are all zero.

Optimality criteria for orders

This section aims to show the specific properties of an ordering that induces an optimal cost. We define the notion of multiset of costs that will help expressing optimality criteria for orders.

Definition 4 (Multiset of a cost) *Given a graph $G = (V, E)$ and an order $\prec$ over V, the multiset of costs $MC(G, \prec)$ is the multiset composed of the values $(|succ_{\prec}(u)| + w(u))$ for each vertex $u \in V$. Its **size** is defined as the number of its elements: $|M| = |V|$. Its **linear cost** is the sum of elements in the multiset: $c_1(M) = \sum_{c \in M} c$. Its **squared cost** (or simply* cost*) is $c_2(M) = \sum_{c \in M} c^2$.*

Property 2 *For a graph G (weighted or not) the size and the linear sum of the multiset $MC(G, \prec)$ do not depend on the order $\prec$. Thus we can note $c_1(G)$ the linear cost of a multiset for G with any order.*

Proof: Take $M = MC(G, \prec)$. By definition, $|M| = |V|$ does not depend on $\prec$. As for the linear cost, $c_1(M) = \sum_{c \in M} c = \sum_{u \in V} |succ_{\prec}(u)| + w(u) = |E| + \sum_{u \in V} w(u)$ does not involve $\prec$ either.

□

Property 3 *When there exists some $d \in \mathbb{N}$ such that $MC(G, \prec)$ contains only the values d and $d + 1$ then the order $\prec$ is optimal.*

Proof: Suppose that a multiset M contains two elements x, y such that $x + 1 < y$. Then consider the multiset M' that is identical to M except that x is replaced by $x + 1$ and y is replaced by $y - 1$. The linear costs of M and M' are equal, but the squared cost of M' is strictly lower: $c_2(M) - c_2(M') = \left(x^2 + y^2\right) - \left((x + 1)^2 + (y - 1)^2\right) = 2\,(y - x - 1) > 0$. So the multiset with lowest squared cost has only values c and $c + 1$. Now consider such a multiset that has k values d and ℓ values $d + 1$. Its linear cost is $c_1(G) = kd + \ell(d + 1) =$

$(k+\ell)d+\ell = |V|d+\ell$. Necessarily, d is the quotient of the Euclidean division of the linear cost: $d = \lfloor \frac{c_1(G)}{|V|} \rfloor$. Then ℓ is the remainder of this division, and k is given by $|V| - \ell$. Thus, the values d, k, ℓ are uniquely determined by G, independently of the order $\prec$.

□

While the property above is true for any graph (weighted or not) it is not relevant for weightless graphs: in a weightless graph, the last vertex u of the elimination order $\prec$ has $|succ_\prec(u)| = 0$, and more generally the vertex u_i that is ordered in the i-th position from the end has $|succ_\prec(u_i)| < i$. The following property handles this case:

Property 4 *For a weightless graph, when there exists $d \in \mathbb{N}$ such that $MC(G, \prec)$ contains all integers from 0 to $d+1$ and at most once the integers 0 to $d-1$, then $\prec$ is an optimal order.*

Proof: The proof of optimality is similar to the proof of property 3: when the property does not hold, we can find two elements x and y with $x + 1 < y$ and we can diminish the squared cost by replacing x by $x + 1$ and y by $y - 1$.

□

Finally, let us introduce the notion of *marginal cost* for a multiset, to measure how much a multiset deviates from the optimal repartition (as given in property 3).

Definition 5 (Marginal cost) *Given a multiset M of size n, we can compute d such that the linear cost of M is $n \times d + v$ where $1 \leq v \leq n$. The **marginal cost** c_m of M is defined as:*

$$c_m(M) = \sum_{u \in M} \max\Big(0, u - (d+1)\Big)$$

Note that we can equivalently define the marginal cost of M as $c_1(M) - \sum_{u \in M} min(d+1, u)$. We know from property 3 that the multiset that minimises the cost with the same linear cost and the same size only contains d and $d+1$ (with at least one $d+1$ since $v > 0$). In other words the marginal cost counts the excess of values over the average cost: if the multiset has an element $d+2$, its marginal cost increases by 1; if it has a $d+5$, the marginal cost increases by 4, etc.

Note that the marginal cost cannot be used directly to decide if an order is optimal. Indeed, consider the two following multisets: M composed of nine times the value 10 and one time the value 11 and M' composed of nine times the values 11 and one time the value 2. They have the same size, the same linear cost and the same marginal cost (which is 0) but M has a lower squared cost than M'.

The following property describes the minimal cost among all the multisets with the same size, the same linear cost and the same marginal cost:

Property 5 *Among all the multisets that have a size n, a linear cost of $d \times n + v$ with $1 \leq v \leq n$ and a marginal cost of at least k (with $2k < v$), then the ones achieving the minimal squared cost are composed of k times the value $d+2$, $v - 2k$ times the value $d+1$ and $(n - v + k)$ times the value d.*

Proof: Take such a multiset M with size n, linear cost of $d \times n + v$ and marginal cost of at least k, satisfying $2k < v$. Suppose that M contains a value $d - i$ with $i > 0$. There are two cases: if M contains a value $d+1$, then replacing $d - i$ by $d - i + 1$ and $d + 1$ by d reduces the squared cost and does not change the marginal cost $c_m(M)$.

Otherwise, M contains a value $d + j$ with $j > 0$ since the linear cost of M is strictly larger than $d \times n$. By definition of the marginal cost, M contains at most c_m elements that

are larger than $d+1$; all other elements are at most d with at least one at $d-i<d$. So, we have $\sum_{u\in M} min(d+1,u) < nd + c_m$. Using $c_m(M) = c_1(M) - \sum_{u\in M} min(d+1,u)$, we find that

$$c_m > nd + v - (nd + c_m) \Rightarrow 2c_m > v$$

By assumption, $v > 2k$, so $c_m > k$. Consequently, replacing $d-i$ by $d-i+1$ and $d+j$ by $d+j-1$ reduces the squared cost while keeping the marginal cost above k. Hence the fact that the values of a multiset with minimal squared cost are greater or equal to d.

□

This property can then be used to compare multisets and is summarised by the following property:

Property 6 *When M has a marginal cost of k then the cost of M is at least $2k$ larger than the balanced distribution (as given by property 3). This $2k$ bound is reached for the optimality criterion described in property 5.*

Proof: As seen before the optimal can be reached by taking two values $x, y \in M$ with $x+2 \leq y$ and changing them to $x+1$ and $y-1$. This balancing operation can reduce by at most 1 the marginal cost but reduces the squared cost by $2(y-x-1) \geq 2$, which is exactly 2 when $x+2 = y$. Starting from a marginal cost of k, reaching the optimal requires at least k such balancing operations, which goes with a reduction of at least $2k$ of the squared cost. Notice that when dealing with an optimal multiset in the sense of property 5 we only combine a d with a $d+2$ which gives us the exact bound. Conversely if we are not in the case of 5 we will have to combine something below (or equal to) d with something larger than $d+3$ or do a combination that does not diminish the marginal cost (such as combining $d+1$ and $d+3$).

□

Reducing weighted-C^{++} to C^{++}

Any instance of C^{++} can be seen as an instance of weighted-C^{++} where the weights are set to zero. On the other hand, this section proves that solving C^{++} is as easy as solving weighted-C^{++} in terms of complexity classes. We propose a polynomial reduction where the cost $w(u)$ of a vertex u is converted into $w(u)$ new successors of u. To ensure that these new neighbours are indeed successors for any optimal order, they are attached to a gadget. The family of gadgets L_d is depicted on Figure 5.4 and defined as:

Definition 6 (L_d family of graphs) *For $d \in \mathbb{N}$, the graph L_d consists of $2(d+1)$ vertices e^d, $v_1^d \dots v_d^d$, $K_0^d \dots K_d^d$ such that $K_0^d \dots K_d^d$ form a clique of $d+1$ vertices, and each vertex V_i^d is connected to e^d and to each vertex K_j^d.*

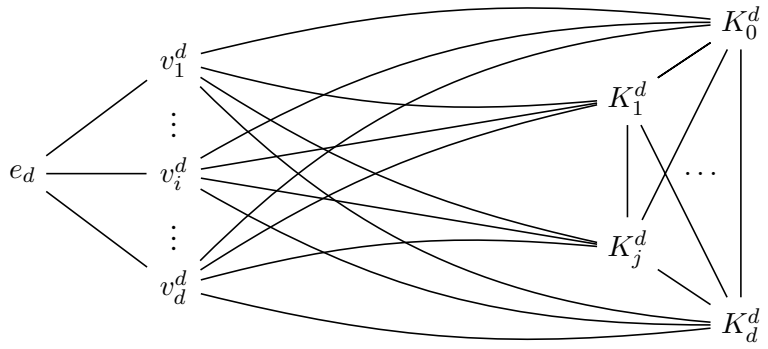


Figure 5.4: **Gadget L_d used to convert vertex weights into new successors.**

Property 7 (Minimum cost for L_d) *In the weightless case, the minimum cost C_d for L_d is induced by the order that starts with e^d followed by the v_i^d vertices and finally by the K_i^d vertices.*

Proof: With this order, the cost is d^2 for e^d, $(d+1)^2$ for each v_i^d and i^2 for K_i^d (supposing we start with K_d^d and end with K_0^d). This is optimal by virtue of property 4.

□

Property 8 (Minimum cost for L_d with a weight 1 on e^d) *In the weighted case, suppose that e^d has weight 1 and the other vertices have weight 0. Then the minimum cost C_d for L_d is induced by the order that starts with e_d followed by the v_i^d vertices and finally by the K_i^d vertices.*

Proof: Let us prove that there is an optimal order starting with e^d. For that, consider any order $\prec$ and let us show that $\prec$ can always can be improved to an order that places the vertex e^d in first position. The order $\prec$ ranks three types of vertices: e^d, V vertices and K vertices, according to the description above. Let us first suppose that there is a vertex of type K before a vertex of type V, itself before the single vertex of type E. In that case the first i vertices are of type V (we can have $i = 0$), then we have $j + 1$ vertices of type K and then one vertex of type V. We note this sequence V^iK^jKV. Let us compute the cost variation ΔC if we exchange this last K with this last V to obtain a sequence V^iK^jVK. The cost only changes for the V and K: it increases from $(d-j)^2$ to $(d-j+1)^2$ for V, whereas it decreases from $(2d-i-j)^2$ to $(2d-i-j-1)^2$ for K. Overall,

$$\begin{aligned}\Delta C &= \Big((2d-i-j)^2 - (2d-i-j-1)^2\Big) + \Big((d-j)^2 - (d-j+1)^2\Big)\\ &= 2(d-i-2)\end{aligned}$$

Therefore, unless $i = d-1$, the cost decreases which means that we can always move the V vertices at the beginning except for maybe one to reduce the cost of the order. In the end we have that the beginning of an optimal sequence can be restricted to the form V^iK^lE or $V^{d-1}K^lVE$. In the first case, transforming V^iK^lE into EV^iK^l decreases the score by $i^2 + i \geq 0$. In the second case, transforming $V^{d-1}K^lVE$ into $EV^{d-1}K^lV$ decreases the score by $d^2 + d - 2l \geq 0$ since $l \leq d$. Thus, in all cases, moving e^d to the beginning of the order reduces the related cost.

We have proved that the minimum cost can be achieved by placing e^d at the beginning of the order. The additional weight on e^d causes a $2d + 1$ cost, but the rest of the order

is not affected by the cost. Starting from the minimum cost C_d of the weightless case of property 7, the minimum cost for weight 1 on e^d is therefore $C_d + 2d + 1$, achieved with the same order.

□

The fact that minimum costs are obtained when e^d is at the beginning of the order in L_d means that L_d is a gadget that forces e^d to have d successors. Before using this property to bridge C^{++} to weighted-C^{++}, let us see how several gadgets interact together.

Let us consider a graph G composed of two subgraphs G_1 and G_2 plus exactly one edge $\{e_1, e_2\}$ with $e_1 \in V_1$ and $e_2 \in V_2$. The cost induced by an order $\prec$ on V only depends on the order over V_1, the order over V_2, and the order between e_1 and e_2. Therefore, an optimal order for G can be seen as either an optimal for G_1 and an optimal order for G_2 where we add a weight of 1 on e_2 (if $e_2 \prec e_1$), or an optimal order for G_2 and an optimal order for G_1 where we add a weight of 1 on e_1 (if $e_1 \prec e_2$). As a result, we obtain the following property:

Property 9 (Cost of a partition) *If adding a weight 1 on e_1 in G_1 increases the minimum cost of G_1 by x and if adding a weight 1 on e_2 increases the minimum cost of G_2 by $y \leq x$, then the minimum cost of G is equal to the minimum cost of G_1 plus the minimum of G_2 where we add a weight of 1 on e_2.*

Using the L_d gadget and the property of the cost of a partition, we will show that there is a polynomial reduction from weighted-C^{++} to C^{++}. Thus, if the weighted-C^{++} problem is strongly NP-hard, then the C^{++} problem is also NP-hard.

Property 10 *Let (G, K) be an instance of the weighted problem, we can compute an equivalent instance of the weightless problem in a time polynomial in the number of edges and vertices in G plus the sum of weights in G.*

Proof: If all the weights in G are zeros, the instance of weighted-C^{++} is already an instance of C^{++}. Suppose that there is a vertex u with a weight $w > 0$ and a degree $d - w$. Let us create the graph G' as a copy of G where u has weight $w - 1$ and is linked to the vertex e^d of a L_d graph. We claim that the minimum cost of G' is lower than $K + C_d$ if and only if the minimum cost of G is lower than K.

Recall that, by property 8, adding a weight 1 on e^d increases the minimum cost of the graph L_d from C_d to $C_d + 2d + 1$. Besides, the sum of the degree of u plus its weight is d, so adding a weight 1 on u increases the cost of any order $\prec$ by at most $2d + 1$. Now let us apply property 9 with $G_2 = G$ and $G_1 = L_d$. Adding a weight of 1 to one of the handles $e_2 = u$ and $e_1 = e^d$ gives a maximum cost increase $x = 2d + 1$: the minimum cost of G' is the minimum cost of G where vertex u has a weight increased by 1. In other words, it is equivalent in terms of minimum cost to handle the graph G or to handle the graph G' where the weight of vertex u has been decreased by 1 unit.

By applying $\sum_u w(u)$ times this property we obtain an instance (G^*, K^*) of the weightless problem which is equivalent to the instance of the weighted problem (G, K). This new instance has $\sum_u w(u) \times |L_{d_u + w(u)}|$ more vertices than the original one, which is polynomial in the size of G plus the sum of weights, and the resulting instance can be computed in polynomial time.

□

Reducing Set Cover to weighted-C^{++}

Our reduction for the weighted case will be a *strong reduction*, meaning the version of the problem where the weights are polynomial in the size of the graph is still NP-hard. It will be based on the *Set Cover problem*, a NP-complete problem that figures among the famous list by Karp [1972]. We recall here the definition of this problem:

Problem 5 (Set Cover) *Given two integers n, k, we denote U the set of elements $\{1, \dots, n\}$. Let P be a set of sets of elements of U, does there exist a subset $P' \subset P$ of size k such that $\cup_{S \in P'} S = U$?*

Let us fix an instance $SC = (P, n, k)$ of the Set Cover problem asking whether we can find k sets $S_1, \dots, S_k$ in P such that $S_1 \cup \dots \cup S_k = U$. We suppose, without loss of generality, that the instance is not trivial in the sense that $|P| \geq k$ (there are at least k sets in P), $\cup_{S \in P} S = U$ (each integer in U is contained in at least one $S \in P$) and that all sets $S \in P$ are such that $S \subseteq U$.

Let us exhibit a weighted graph G and a value V such that minimum cost for G is less than V if and only if SC is feasible, meaning that $\{1, \dots, n\}$ can be covered with k sets from P. This graph is depicted on Figure 5.5 and defined as:

Definition 7 (Weighted graph $G(SC)$ of a Set Cover instance) *Given a Set Cover instance $SC = (P, n, k)$, let us construct the graph $G(SC)$ with:*

- *vertices: a special vertex A, n vertices $e_1, \dots, e_n$ for the elements of U, ℓ vertices $s_1, \dots, s_\ell$ for the sets of P, and three vertices a^i_j, b^i_j, c^i_j for each $i \in S_j$.*
- *edges: A is linked to all vertices of the form s_j or e_i; for a pair (i, j) with $i \in S_j$, both a^i_j and b^i_j have an edge with s_j and c^i_j; in turn c^i_j has an edge with e_i.*
- *weights: for some parameter d, $w(A) = d + 1 + k - \ell$ and $w(c^i_j) = d$, while each other vertex u has weight $d + 2 - d_u$.*

Parameter d. The weights of vertices are given so that the cost of each vertex reaches a specific value. Recall that the cost of a vertex is the sum of its degree and its weight. In $G(SC)$, we want the cost of each vertex to be $d + 2$, except for the c^i_j that have a cost of $d+3$, and for A that has a cost of $d+1+n+k$. Parameter d can be any integer, but it needs to be large enough so that all weights are positive. This is not constraining for vertices a^i_j, b^i_j and c^i_j, as their degree is fixed. Vertex s_j is linked to A and to a^i_j and b^i_j for each element i in S_j, so its degree is $1 + 2 \times |S_j|$: to ensure that $w(s_j) \geq 0$, it suffices that $d > 2 \times |S_j|$. Vertex e_i is linked to A and to c^i_j for each set S_j that contains i, so its degree is at most $1 + \ell$. Vertex A has degree $\ell + n$. Having the additional condition $d > \ell$ is sufficient to guarantee the constraint on A and e_i vertices.

Value of V. As we will show, when there is a Set Cover with k sets then we have an order $\prec$ for G such that $MC(G, \prec)$ contains k times the value $d + 2$ (corresponding to the k selected sets), $\sum_{S \in P} |S| - n$ times the value d and all the other values are $d + 1$. It implies that the cost $V = k(d+2)^2 + (\sum_{S \in P} |S| - n)d^2 + r(d+1)^2$, where r is the number of vertices in G minus k and minus $(\sum_{S \in P} |S| - n)$.

Note that, per property 5, this value V corresponds to the minimal cost for an order that has a marginal cost of k. Conversely, we will show that if there is a solution with a marginal cost of k or less then there is a Set Cover with k sets, proving that it is a reduction.

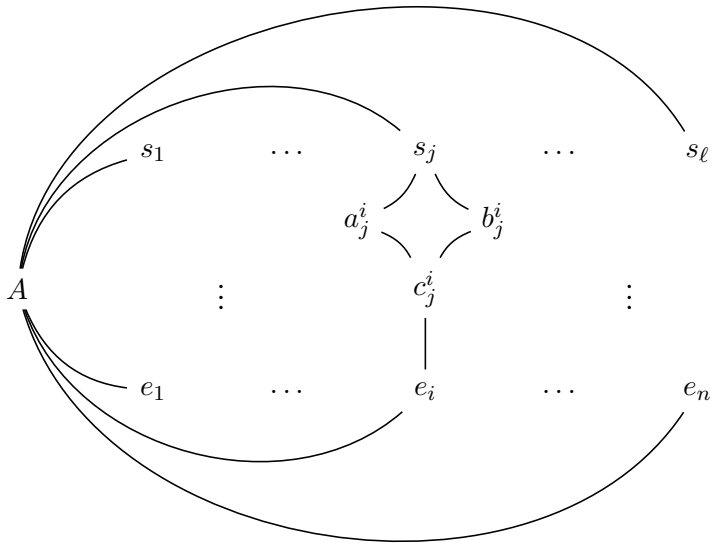


Figure 5.5: **An instance SC of Set Cover is converted to this graph $G(SC)$.**

Note that this converse direction is stronger than what is needed as there exists multisets with a marginal cost of k that do not match the minimal cost.

The general intuition underlying the equivalence between a solution (if any) of the Set Cover problem and a solution of the corresponding weighted-C^{++} problem is the following. The first k vertices s_j selected in the elimination order correspond to the S_j sets that cover U. Indeed, each of these vertices generates exactly a marginal cost of 1 and all other vertices according to the elimination order will not generate any marginal cost if we can eliminate all e_i vertices without adding any marginal cost. This condition is met if deleting the k first s_j vertices allows to decrease the cost of all e_i vertices by (at least) 1 unit, which means that we have deleted at least one triplet a_j^i, b_j^i, c_j^i related to vertex e_i. If so, we have found an elimination order with cost V as well as k sets $S_1 \dots S_k \in P$ which cover U.

Proposition 3 (solution to Set Cover $\Longrightarrow$ solution to weighted-C^{++}) *Given an instance SC of Set Cover and the associated graph $G(SC)$, if all the elements of U can be covered with at most k sets then there exists an order $\prec$ on the vertices of $G(SC)$ that has a cost V.*

Proof: Suppose that we have a solution to Set Cover with the sets $S_{j_1}, \dots, S_{j_k}$. Let us prove that our graph G has an elimination order where the cost of each vertex is d or $d+1$ or $d+2$ but with only k vertices with cost $d+2$.

The elimination order can be built by having j going through $j_1, \dots, j_k$. For each j value, we eliminate first s_j for a cost of $d+2$, then we go through $i \in S_j$ and eliminate the corresponding a_j^i and b_j^i vertices (both at cost $d+1$ once s_j has been removed). Then we eliminate c_j^i (for a cost of d if e_i is already eliminated and $d+1$ otherwise). Finally, if e_i has not yet been eliminated by a previous j value, we eliminate it for a cost of $d+1$.

After this, the vertex A has lost $k+n$ neighbors: all the e_i and the k vertices s_j were selected. Its remaining cost is $d+1$ so we eliminate it, which in turn means that all the remaining s_j have a cost of $d+1$ and we can eliminate them all (with their a_j^i, b_j^i and c_j^i attached).

Overall the cost of this elimination order is exactly V.

□

Proposition 4 (solution to Set Cover $\Longleftarrow$ solution to weighted-C^{++}) *Given an instance SC of Set Cover and the associated graph $G(SC)$, if there exists an order $\prec$ on the vertices of $G(SC)$ that has a cost at most V, then all the elements of U can be covered with at most k sets.*

Proof: Suppose that we have an order $\prec$ such that the total cost is below V. Since V is the optimal cost for a marginal cost of k, by property 6 the marginal cost of $\prec$ is at most k. We now extract a solution to the corresponding Set Cover instance.

First we notice that when A is eliminated, its cost is $d+1+k-E_s+E_n+R_n$ where E_s the number of s_i eliminated, and R_n is the number of e_i remaining. However, as long as A is not eliminated, E_s is less than (or equal to) the marginal cost of all the vertices eliminated before A. Indeed, if s_j is eliminated while A is still present it is because we have paid a marginal cost at least 1 to eliminate directly one of s_j or a_j^i or b_j^i or c_j^i for $i \in S_j$. That is true because if A is present, then all those vertices have a cost of $d+2$ (except c_j^i that has a cost $d+3$ if e_i is not eliminated yet).

Overall when we eliminate A, we pay a marginal cost of $(k-E_s)+R_n$. The marginal cost of $\prec$ is at least the marginal cost of all vertices eliminated before A (which is at least E_s) plus the marginal cost for A. Hence a marginal cost larger than $E_s+(k-E_s)+R_n = k+R_n$. Assuming this marginal cost is k means that $R_n = 0$; in other words, all vertices e_i corresponding to integers $\{1 \dots n\}$ have been eliminated before A.

Note that if a vertex e_i is directly eliminated without eliminating first a vertex s_j and a triplet a_j^i, b_j^i, c_j^i then we have to add a marginal cost of 1 specifically for this vertex e_i. But in that case, it means that the marginal cost of all vertices before A includes the cost of removing this e_i which means that we cannot have an overall marginal cost of k.

Combining everything we get that if we have an order that has a marginal cost of k and thus a cost of at most V, then we have k sets $S_{i_1}, \dots, S_{i_k}$ covering all integers in U.

□

We proved that Set Cover is NP-complete, that Set Cover reduces to weighted-C^{++}, and that weighted-C^{++} reduces to C^{++}. In conclusion, C^{++} is NP-complete in its decision version; the minimisation version is therefore NP-hard.

5.3.3 Relaxations, bounds and approximations

Relaxing the constraint of acyclic orientation

While we focused until here on node orderings, the complexities of Algorithms 2 and 3 depend more precisely on an edge orientation. Node ordering is one way to orient the edges, from low to high indices. The resulting directed graph is of a particular type: it is acyclic, which means that there is no directed path that starts and ends on the same node. This absence of cycles allows the algorithms to avoid redundancy while listing triangles, but any orientation without cyclic triangles would have the same effect, regardless of the existence of longer cycles. It is thus interesting to relax the acyclic constraint and to know what happens to the C^{++} and C^{+-} problems when they are considered as minimisation

problems over the full set of edge-orientations. Let us refine their definition so that they adapt to any edge-orientation.

Definition 8 (Cost induced by an edge-orientation) *Given an undirected graph G, an edge-orientation is a function $\eta : E \to V^2$ that indicates for each undirected edge $\{u, v\} \in E$ a directed edge (u, v). The set of directed edges is noted E_η. The costs C^{++} and C^{+-} induced by an orientation η of the edges are defined by:*

$$C^{++}(\eta) = \sum_{(u,w)\in E_\eta} d_u^+ \qquad C^{+-}(\eta) = \sum_{(u,v)\in E_\eta} d_v^+$$

With this generalisation of the C^{++} and C^{+-} problems, more open problems arise. Table 5.1 shows the known results for both problems and both types of constraint on the edge-orientations; it also sums up approximability results from the literature. Each of these additional results are discussed below, and their demonstrations are briefly presented.

	Orientation without cycle	**Arbitrary orientation**
Cost C^{++}	NP-hard (Section 5.3.2) There exists a 2-approximation	Polynomial
Cost C^{+-}	NP-hard (Section 5.3.1) Best known approximation $O(\sqrt{\log n})$	

Table 5.1: **Complexity and approximability of C^{++} and C^{+-}** minimisation problems depending on the constraint of edge-orientations.

Additional results for C^{+-}

C^{+-} is NP-hard for any edge-orientation. The proof of NP-hardness for C^{+-} presented in Section 5.3.1 is based on a reduction from a satisfiability problem. From a specific instance of NAE3SAT+, it creates a graph where the minimum C^{+-} value is as low as a threshold if and only if the instance is satisfiable.

Interestingly, the minimum cost obtained among node orderings is also minimum among all the edge-orientations. Indeed, the minimum C^{+-} cost for a satisfiable instance of m clauses is $2m$. This value comes from the triangle of Figure 5.3 that exists for each clause gadget: a triangle $\{u, v, w\}$ where each node has an additional neighbour. For such a pattern, any node ordering π gives a C^{+-} cost at least 2: assume $\pi_u < \pi_v < \pi_w$, then u may have no predecessor and w may have no successor, but v has one of each plus its additional neighbour: $C^{+-}(\pi) \geq d_v^+ \cdot d_v^- = 2$. General edge-orientations would allow for one other situation η, where the edges of the triangle are a cycle $(u, v), (v, w), (w, u)$. In this case, each node has exactly one predecessor and one successor plus its additional neighbour, which leads to $C^{+-}(\eta) = 6$ for each clause (and possibly extra cost for the literals).

Thus, minimising C^{+-} is NP-hard, whether over the set of edge-orientations or the set of node orderings (or even the set of edge-orientations without 3-cycle).

C^{+-} can be as low as zero. While trying to minimise C^{+-} is hard on general graphs, it is straightforward if they have a certain property: on bipartite graphs, there is a node ordering such that the C^{+-} value is zero. Consider a bipartite graph $G = (A \cup B, E)$ such

that edges are only between A and B. Define an order $\prec$ such that $a \prec b$ for any $a \in A$ and $b \in B$ (for instance $a_1 \prec \cdots \prec a_{|A|} \prec b_1 \prec \cdots \prec b_{|B|}$). Then any $a \in A$ has $d_a^- = 0$ so it does not contribute to C^{+-}. Similarly, for all $b \in B, d_b^+ = 0$. In total, $C^{+-}(\prec) = 0$.

C^{+-} does not have constant-factor approximations. On a scientific forum, Makarychev [2020] shows that C^{+-} is linked to the Max-Cut problem and to its complementary Min-Uncut. All three problems intuitively describe how far from bipartite a graph is. Indeed, when the graph is bipartite, we saw just above that C^{+-} can be zero; the maximum cut of such a graph contains all the edges, and the minimum uncut contains no edge at all.

Using this tight link Makarychev shows that the following properties, that have been previously proven by Agarwal et al. [2005] for the Min-Uncut problem, also apply to C^{+-}. First, there exists a polynomial time $O(\sqrt{\log n})$-approximation. Second, it cannot be approximated in polynomial time within a constant factor, unless the Unique Game Conjecture of Khot [2002] is wrong.

This strong statement of inapproximability is an incentive to design heuristics to address the C^{+-} problem instead of using algorithms with mathematical guarantees. Different heuristics are presented in Section 5.4 and tested in Section 5.5.

Additional results for C^{++}

C^{++} is in P for general edge-orientations. The proof of Section 5.3.2 shows that C^{++} is NP-hard to minimise over the set of vertex orderings. However, Young and Jachiet [2020] have given independent proofs that the minimisation over the set of edge-orientations is polynomial.

First of all, a naive greedy algorithm does not reach the minimum solution. Starting from any edge-orientation, such an algorithm takes an edge and inverts it if it reduces the total C^{++} cost. Yet, the counter example of Figure 5.6 shows a situation where C^{++} is not minimum, but no single edge inversion reduces it. In that example, the way to obtain a lower C^{++} is to invert a path of three edges: the path inversion only impacts the starting and ending nodes, whose contribution goes from 0 to 1 and from 4 to 1 respectively. This hints at a first type of polynomial algorithm.

The method of Jachiet to reach the minimum C^{++} value is a path inversion. If we can find a path whose starting node has a higher out-degree than the ending node, the path is inverted and C^{++} decreases. Intermediate nodes of the path are not impacted, as they gain one predecessor on one side and lose it on the other. An interesting proof then shows that in the absence of invertible path, C^{++} is minimum, thus sketching a polynomial time algorithm for the optimisation over edge-orientations.

The main method of Young consists in reducing the problem to the minimum-weight bipartite matching problem, which is known to be polynomial [Kleinberg and Tardos, 2006]. The reduction starts from an instance of C^{++} and creates a bipartite graph with specific weights on the edges. The proof then shows that finding a matching of minimum weight in this new graph also yields an edge-orientation with minimum C^{++} in the initial graph.

Finally, Young presents sketches of proofs with linear programming and randomised rounding methods. They consist in reformulating the C^{++} problem in a set of variables with constraints, and then using the polynomial time algorithms for such standardised problems.

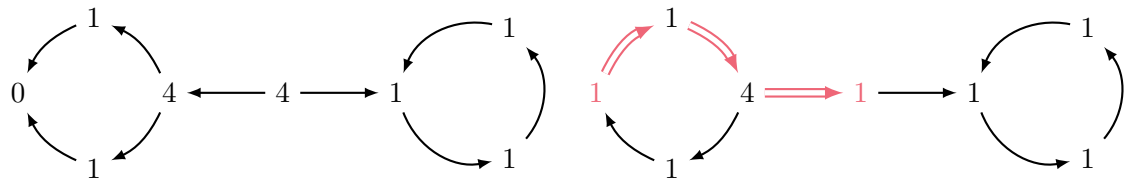


Figure 5.6: **Counter example for the greedy algorithm of C^{++} over edge-orientations.** The numbers indicate the contribution of each node u to the C^{++} cost, which is $(d_u^+)^2$. On the left, the cost is 13 and cannot be lowered by any single edge inversion. On the right, the simultaneous reversion of the three double-edges gives an orientation with $C^{++} = 11$.

Lower-bounds exist for C^{++}. The C^{++} problem can be reformulated as a minimisation of $S = \sum_{i=1}^{n} x_i^2$ where $x_1, \dots, x_n$ are integers of sum m. According to Jensen's inequality, S is minimised if $x_1 = \cdots = x_n = \frac{m}{n}$, in which case $S = \frac{m^2}{n}$. This gives a first theoretical lower-bound for C^{++}, that is minimised if all nodes have the same out-degree: any ordering π satisfies $C^{++}(\pi) \geq \frac{m^2}{n}$. Further refinements are possible, as $\frac{m}{n}$ is not always a correct value for the out-degree. First, $\frac{m}{n}$ may not be an integer. If we note $m = qn + r$ with $0 < r < n$, then the minimum is achieved if r nodes have out-degree $q+1$ and the others have out-degree q: $C^{++}(\pi) \geq (n-r)q^2 + r(q+1)^2 = \frac{m^2}{n} + r(1 - \frac{r}{n})$. Second, some nodes cannot have an out-degree as high as q, either because they are at the end of the ordering (the i-th last node has out-degree at most $i-1$), or because their total degree is lower. This leads to Algorithm 4 that computes a lower-bound for C^{++}.

Algorithm 4 – Computing a lower-bound for C^{++}

Input: graph $G(V, E)$, ascending degree ordering δ

$n \leftarrow |V|$, $m \leftarrow |E|$
for each vertex u in order δ_u **do**
    $x_u = \min(u, d_u, \lfloor \frac{m}{n} \rfloor)$
    $n \leftarrow n - 1$
    $m \leftarrow m - x_u$
return $S = \sum_{u \in V} x_u^2$

Core ordering is a 2-approximation of C^{++}. Core ordering is one of the standard node orderings used in the literature of pattern mining, in particular for triangles, as explained in Section 5.2. In their work on triangle sampling, Turk and Turkoglu [2019] are interested to minimise the cost $C(\pi) = \sum_{u \in V} \binom{d_u^+}{2}$ over node orderings π. This quantity is related to C^{++} in the following way:

$$C(\pi) = \frac{C^{++}(\pi) - m}{2} \qquad \text{or} \qquad C^{++}(\pi) = 2 \cdot C(\pi) + m$$

This formula indicates that an order π^* that minimises C also minimises C^{++}. Turk and Turkoglu show in Theorem 2 that the core ordering γ is a 2-approximation of such an optimum ordering: $C(\gamma) \leq 2 \cdot C(\pi^*)$. This result directly translates for C^{++}:

$$C^{++}(\gamma) = 2 \cdot C(\gamma) + m \leq 4 \cdot C(\pi^*) + m \leq 2\Big(2 \cdot C(\pi^*) + m\Big) = 2 \cdot C^{++}(\pi^*)$$

Hence the fact that the core ordering is a 2-approximation for C^{++}.

Ordering strategy	Chosen algorithm	
	A++ (cost C^{++})	A+− (cost C^{+-})
Reducing C^{++}	Degree or core ordering Ortmann and Brandes [2013]	Core ordering Danisch et al. [2018]
Reducing C^{+-}	–	our contributions Section 5.4.3

Table 5.2: **Combination of orderings and algorithms.** The literature misses combinations of A+− algorithms with ordering strategies that specifically reduce C^{+-}.

5.4 New orderings to reduce the cost of triangle listing

In this section, we discuss how to design vertex orderings to reduce the cost of triangle listing algorithms. Recall that for an ordering π, the costs of interest are $C^{++}(\pi) = \sum_{u \in V} d_u^+ d_u^+$ and $C^{+-}(\pi) = \sum_{u \in V} d_u^+ d_u^-$. They correspond to the influence of node ordering on the number of operations of Algorithms 2 (A++) and 3 (A+−) respectively.

The fastest triangle listing methods identified in the literature present an inconsistency between the choice of ordering and the choice of algorithm, as shown in Table 5.2. In the experiments of Ortmann and Brandes [2013], the fastest method identified is to run algorithm A++ with core or degree ordering. For Danisch et al. [2018], the fastest is algorithm A+− with core ordering. The intuition behind both orderings is that high degree vertices are ranked after most of their neighbours in π so that their out-degree in G_π is lower. This reduces the cost C^{++}, which in turn reduces the number of operations required to list all the triangles as well as the actual running time of A++. The fact that A+− also performs well with core ordering is considered as a side effect by Ortmann and Brandes.

This chapter aims at bridging the gap in Table 5.2. To our knowledge, no previous work has designed orderings with a low C^{+-} cost and used them with A+−. As Theorem 1 of Section 5.3 proves that finding an ordering that minimises C^{+-} is NP-hard, this section will focus on efficient heuristics to reduce the C^{+-} cost. The experiments of Section 5.5 will then show that such orderings can lower the computational cost further than the usual degree and core orderings.

5.4.1 Comparing the costs C^{+-} and C^{++}

While the formulas of C^{+-} and C^{++} are similar, their intuitive meanings are different. To reduce C^{++}, the goal is to find an ordering where the out-degree is uniform across all nodes: the non-decreasing degree ordering will place nodes with low total degree at the beginning so that all their neighbours are successors; the high-degree nodes will be towards the end of the ordering to ensure that most of their neighbours are predecessors. To reduce C^{+-}, the goal is to find an ordering where each node has most of its neighbours on only one side, either predecessors or successors.

Several facts hint that, on a given graph, C^{+-} may reach lower values than C^{++}. First, the fast implementation provided by Danisch et al. [2018] suggests that A+− algorithm can be faster that A++ even when the ordering is designed to reduce C^{++}; it could further accelerate with an ordering that reduces C^{+-}. Second, all bipartite graphs have an ordering where $C^{+-} = 0$, while any graph with m edges has $C^{++} \geq m$. Finally, if we consider a fully connected graph, C^{+-} is smaller than C^{++} regardless of the ordering: in a triangle for instance, any ordering gives an out-going degree distribution $(0, 1, 2)$, so $C^{++} = 0^2 + 1^2 + 2^2 = 5$ and $C^{+-} = 0 \times 2 + 1 \times 1 + 2 \times 0 = 1$. Table 5.3 indicates the costs

Graph	Cost C^{++}	Cost C^{+-}
Edge	1	0
Triangle	5	1
4-clique	14	4
k-clique	$\frac{1}{6}k(k-1)(2k-1)$	$\frac{1}{6}k(k-1)(k-2)$
$k \to \infty$	$\sim k^3/3$	$\sim k^3/6$

Table 5.3: **Values of C^{++} and C^{+-} in a fully connected graph** for any ordering.

for bigger cliques and shows that the asymptotic cost for cliques is twice as high for C^{++} as for C^{+-}.

The advantage of C^{+-} over C^{++} is also visible in a random graph model. Let us compute their expected values over the set of Erdős and Rényi [1960] graphs, noted $ER(n,p)$ for graphs of n nodes where each edge exists with probability p.

Proposition 5 (C^{++} and C^{+-} in random graphs) *In a $ER(n,p)$ random graph, for a random ordering π, the cost C^{++} is more than twice the cost C^{+-} in asymptotic expectation:*

$$\mathbb{E}\left(C^{++}(\pi)\right) \underset{n\to+\infty}{\sim} 2\cdot\mathbb{E}\left(2C^{+-}(\pi)\right) + \mathbb{E}(m)$$

Proof: Consider a graph $ER(n,p)$ with a random ordering π. Without loss of generality, we assume that node $u \in [\![1,\dots,n]\!]$ is in position $\pi_u = u$. Its number of successors s_u follows a binomial law $\mathcal{B}(n-u,p)$: the number of successors is higher if u is at the beginning of the ordering. This law has expectation $\mathbb{E}(s_u) = (n-u)p$ and variance $\mathbb{V}(s_u) = (n-u)p(1-p)$. Hence the following expectation for C^{++}:

$$\begin{aligned}
\mathbb{E}\left(C^{++}(\pi)\right) &= \sum_{u\in V}\mathbb{E}(s_u^2) = \sum_{u\in V}\mathbb{V}(s_u) + \mathbb{E}(s_u)^2 && \text{as } \mathbb{V}(X) = \mathbb{E}(X^2) - \mathbb{E}(X)^2\\
&= \sum_{u=1}^{n}(n-u)p(1-p) + (n-u)^2p^2 && \text{as } u \in [\![1,\dots,n]\!]\\
&= \sum_{v=0}^{n-1} vp(1-p) + v^2p^2 = p(1-p)\sum_{v=0}^{n-1} v + p^2\sum_{v=0}^{n-1} v^2 && \text{with } v = n-u\\
&= p(1-p)\frac{n(n-1)}{2} + p^2\frac{n(n-1)(2n-1)}{6}\\
&= 2p^2\binom{n}{3} + p\binom{n}{2} \underset{n\to+\infty}{\sim} p^2\frac{n^3}{3} + p\frac{n^2}{2} && \text{when } n >> 1
\end{aligned}$$

In large graphs, C^{++} is at least $\frac{1}{3}p^2n^3$, plus the term $p\binom{n}{2}$ that corresponds to the expected number of edges. Similar equations show that it is only $\frac{1}{6}p^2n^3$ for C^{+-}: the expectation of C^{+-} involves the random variable s_u for the successors, and for the predecessors $d_u - s_u$ that follow a law $\mathcal{B}(u-1,p)$. For a given node, these two variables are independent because all edges are drawn independently.

$$
\begin{aligned}
\mathbb{E}\left(C^{+-}(\pi)\right) &= \sum_{u\in V} \mathbb{E}\left(s_u(d_u - s_u)\right) \\
&= \sum_{u\in V} \mathbb{E}(s_u)\cdot\mathbb{E}(d_u - s_u) && \text{as variables are independent} \\
&= \sum_{u=1}^{n}(n-u)p\cdot(u-1)p = \sum_{v=0}^{n-1} p^2 v(n-v-1) && \text{with } v = n-u \\
&= p^2(n-1)\frac{n(n-1)}{2} - p^2\frac{n(n-1)(2n-1)}{6} \\
&= p^2\binom{n}{3} \underset{n\to+\infty}{\sim} p^2\frac{n^3}{6} && \text{when } n >> 1
\end{aligned}
$$

□

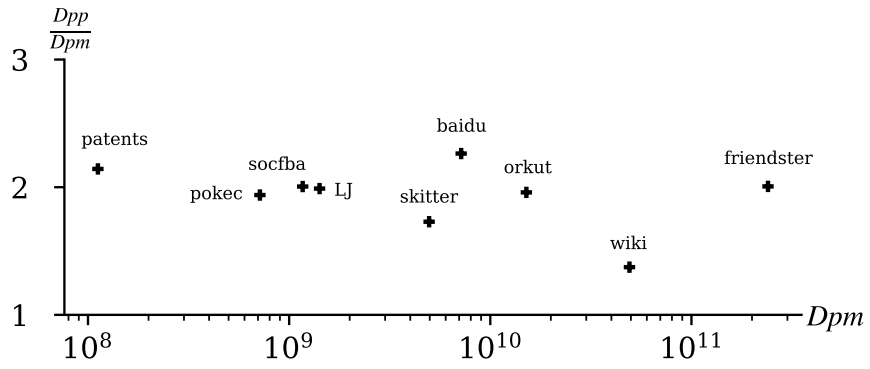


Figure 5.7: **Ratio $\frac{C^{++}}{C^{+-}}$ for 9 datasets with a random ordering.**

As for real-world networks, we also measure the values of C^{++} and C^{+-} obtained for a random ordering on some of the datasets later presented in Table 5.4. Figure 5.7 shows that the C^{+-} cost is always between 1.3 and 2.3 times lower than C^{++}, which relates to the asymptotic limit for cliques reported in Table 5.3: $C^{++} \sim 2C^{+-}$.

These numerous examples could let us think that C^{+-} can always go lower than C^{++}. Let us show that this is not the case, or in other words, that there exists a graph and an ordering π such that any ordering ν satisfies $C^{+-}(\nu) > C^{++}(\pi)$. Figure 5.8 shows one such graph: it comprises a clique of $k = 7$ nodes, each of which is attached to 30 extra nodes of degree one. For C^{++}, let us build π that first orders the extra nodes and then the nodes of the clique. Each extra node has a contribution of 1, and the nodes of the clique together contribute $\frac{1}{6}k(k-1)(2k-1) = 91$, amounting for a total $C^{++}(\pi) = 91 + 7\times 30 = 301$. For C^{+-}, let us build an optimum ordering ν. The extra nodes can go either before or after their unique neighbour without contributing any cost. Each node of the clique can therefore choose how many of its extra neighbours are predecessors and successors. The first node of the clique has 6 successors within the clique and chooses to have 30 extra successors, and thus contributes zero. The second node has one predecessor within the clique and chooses 35 successors, hence a contribution of $1\times(5+30) = 35$. The third node of the clique has two predecessors, 4 successors in the clique and 30 extra successors, which is a contribution of $2\times(4+30) = 68$. The fourth contributes $3\times(3+30) = 99$. The fifth, sixth and seventh nodes of the clique symmetrically contribute 68, 35 and zero. In total, $C^{+-}(\nu) = 305$.

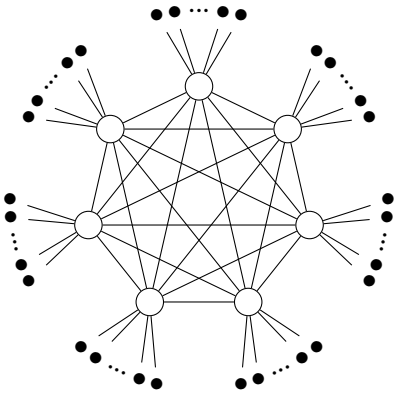


Figure 5.8: **Example of graph where C^{++} can be lower than C^{+-}.** Each of the nodes of a 7-clique are connected to 30 nodes of degree 1 (not all represented on this figure). Placing black nodes before white nodes gives $C^{++} = 301$ while for any ordering $C^{+-} \geq 305$.

All the examples above show that costs C^{+-} and C^{++} have different behaviours depending on the type of graph. The value of C^{+-} is lower in various toy examples as well as random graphs or real-world networks with random ordering. Besides, the current fastest implementation, by Danisch et al. [2018], use the algorithm A+−. These are motivations to investigate orderings that target C^{+-} and to see whether the cost can be reduced beyond what the core or degree orderings achieve.

5.4.2 Distinguishing two tasks for triangle listing

Triangle listing typically consists of the following steps: loading a graph, computing a vertex ordering, and listing the triangles. To determine the speed of a method, it is first necessary to decide which of these steps matter. Unfortunately, the literature does not agree on this matter. On the one hand, time measurements by Latapy [2008], Danisch et al. [2018], Li et al. [2020] only take into account the execution time of the triangle listing algorithm: loading and ordering time are excluded. On the other hand, Schank and Wagner [2005], Ortmann and Brandes [2013] measure the whole execution, starting from an unordered graph in external memory.

To be able to compare our contribution with all the existing literature, we tackle two distinct tasks in our study: we call **mere-listing** the task of listing the triangles of an already loaded graph with a given vertex ordering; we call **full-listing** the task of loading a graph, computing a vertex ordering, and listing its triangles. In the rest of the chapter, we use the notation *task-order-algorithm*: for instance, *mere-core*-A+− refers to the mere-listing task with core ordering and algorithm A+−. Using this notation, the fastest methods identified in the literature are mere-core-A+− for Danisch et al. [2018], mere-core-A++ and full-degree-A++ for Ortmann and Brandes [2013]. We use all three methods as benchmarks in our experiments of Section 5.5.

Studying both tasks gives a better understanding of the phenomena at play in the speed of triangle listing. On the practical side, full-listing represents the runtime for one execution: it favours quickly obtained orderings even if their induced cost is not the lowest. From an algorithmic point of view, mere-listing shows the impact that orderings can have on the cost of triangle listing. As the ordering time is not taken into account, a long time can be spent on finding an ordering with low cost. This time measurement favours situations where the ordering is distributed to other users or used several times. For ex-

ample, a network that is about to be shared to many end users can be ordered once by its owner; there is an incentive to spend significant time in the ordering phase if it reduces the listing time for each end user. Similarly, in a recommendation system that requests graph motif enumeration on the fly, finding an efficient ordering once accelerates each subsequent query. Another situation may arise frequently with large networks: while we limit ourselves to networks that fit in main memory, the list of triangles may be orders of magnitude larger than the list of edges, thus reaching the limit of the memory. With an appropriate ordering, regenerating the list of triangles is fast so it does not need to be stored.

These differences between full-listing and mere-listing indicate that both tasks are worth studying, and they lead to a time-quality trade-off for cost-reducing heuristics: depending on the application, one may be keen on spending more or less time on the ordering phase.

5.4.3 Reducing C^{+-} along a time-quality trade-off

Recall that two efficient algorithms are identified in the literature for triangle listing (Algorithms 2 and 3). Their number of operations respectively depend on the costs C^{++} and C^{+-} induced by the node ordering. However, the orderings that have been considered (degree and core) induce a low C^{++} cost, but not necessarily a low C^{+-} cost.

Our goal here is therefore to design a procedure that takes a graph as input and produces an ordering π with a low induced cost $C^{+-}(\pi)$. In the classification of Chapter 3, this belongs to the mechanism of precedence with an objective function. Because of Theorem 1, finding an optimal solution to this function is not realistic for graphs with millions of edges. Moreover, the time allocated to the ordering phase should be adaptable to the choice of full-listing or mere-listing task. We therefore present three heuristics aiming at reducing the C^{+-} value, exploring the trade-off between quality in terms of C^{+-} and ordering time.

Neigh heuristic

Presentation. We define the *neighbourhood optimisation* method, a greedy reordering where each vertex is placed at the optimal index with respect to its neighbours, as illustrated in Figure 5.9. It is based on the observation that the contribution of a node u

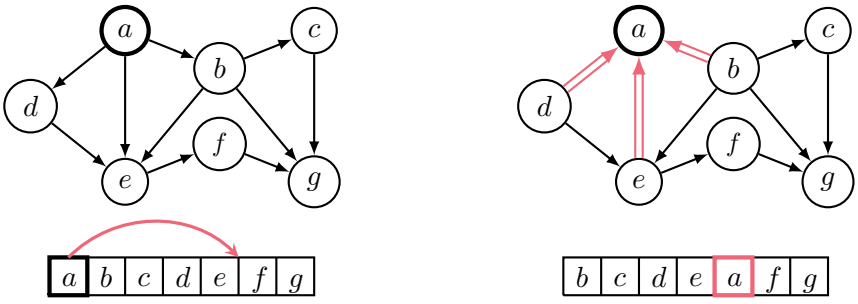


Figure 5.9: **Example of update in the *Neigh* heuristic**: vertex a is moved to a position among its neighbours that induces the lowest cost. The tables shows how the ordering is updated. Edges that changed orientation with the new ordering are represented with double lines. The initial ordering of this example has $C^{+-} = 9$ while the final ordering has $C^{+-} = 6$. The optimal C^{+-} cost for this graph is 3 (with ordering e, g, f, a, c, d, b).

to C^{+-} (or C^{++}) only depends on its relative position to its neighbours. In other words, changing the index π_u only affects $C^{+-}(\pi)$ if the position of u with respect to at least one of its neighbours changes; otherwise the in- and out-degrees of all vertices remain unchanged.

Starting from any ordering π, the algorithm described in Algorithm 5 considers each vertex u one by one (line 3) and, for each possible position $p \in [\![1, d_u]\!]$, it computes $C^{+-}(p)$, the value of C^{+-} when u is just after its p-th neighbour in π, as well as $C^{+-}(0)$ when u is before all its neighbours. The position p_* that induces the lowest value of C^{+-} is selected (line 5) and the ordering is updated (line 6). The process is repeated until C^{+-} reaches a local minimum, or until the relative improvement is under a threshold ε (last line). The resulting π induces a low C^{+-} cost.

Algorithm 5 – Neighbourhood optimisation (*Neigh* heuristic)

Input: graph G, initial ordering π, threshold $\varepsilon \geq 0$

1: **repeat**
2: $\quad C_0 = C^{+-}(\pi)$
3: $\quad$ **for** each vertex u of G **do**
4: $\quad\quad$ sort N_u according to π
5: $\quad\quad$ $p_* = \text{argmin}_{p \in [\![0, d_u]\!]} \{C^{+-}(p)\}$
6: $\quad\quad$ update ordering π to put u in position p_*
7: **while** $C^{+-}(\pi) < (1 - \varepsilon) \cdot C_0$

Complexity. Computing the best position of u among its neighbours is not straightforward. Indeed, summing the d^+d^- values of all the nodes of the neighbourhood and for each position would imply a complexity $\Theta(d_u^2)$, which is equivalent to the complexity of listing the triangles that involve u. Instead, we reach a complexity $\Theta(d_u)$ by first computing $C^{+-}(0)$ and updating it for each successive position. If v is the $(p+1)$-th neighbour of u according to π, then $C^{+-}(p+1)$ only differs from $C^{+-}(p)$ because the contributions of u and v change. By swapping u and v in π, u gains one predecessor and loses one successor, and vice versa for v:

$$\begin{aligned} C^{+-}(p+1) - C^{+-}(p) &= \Big((d_u^+ - 1)(d_u^- + 1) + (d_v^+ + 1)(d_v^- - 1)\Big) - \Big(d^+ud^-u + d^+ud^-u\Big) \\ &= d_u^+ - d_u^- + d_v^- - d_v^+ - 2 \end{aligned}$$

Maintaining the values d^+ and d^- implies a constant time, which gives $\Theta(1)$ operations for each position.

Altogether, each iteration of the *Neigh* heuristic has a time complexity $\Theta(m)$: finding the best position p^* takes $\Theta(d_u)$, and π is updated in constant time using a linked list; adding this over all the nodes gives $\Theta(m + n)$ which is $\Theta(m)$ if there are no isolated nodes. If the improvement threshold ε is reached after I iterations, the overall complexity is $\Theta(Im)$; as we will see in Section 5.5, the process reaches a threshold $\varepsilon = 10^{-2}$ after $I < 10$ iterations on all the networks.

Discussion. This heuristic has several strong points: it is adaptable to other objective functions, for instance C^{++}; it is greedy, so the cost keeps improving until the process stops; if the initial ordering already induces a low C^{+-} cost, the heuristic can only improve it; it is stable in practice, which means that starting from several random orderings give similar final costs;and we show in Section 5.5 that it allows for the fastest mere-listing.

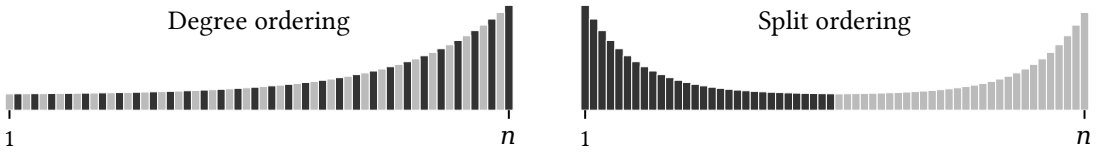


Figure 5.10: **Illustration of *Degree* and *Split* orderings.** Each bar represents a node, the height corresponds to its degree and the colour to its parity. On the left, nodes are ordered according to their degree. On the right, nodes split into two groups so that high degrees are at the beginning or at the end of the ordering.

The *Neigh* heuristic also has an important downside when it comes to the full-listing task: in spite of its linear complexity, it can in practice take longer than the triangle listing step. We therefore propose the following faster heuristics for the full-listing task.

Check heuristic

This heuristic is inspired by core ordering, where vertices are repeatedly selected according to their current degree [Batagelj and Zaveršnik, 2003]. Its mechanisms are precedence and (degree) centrality, with a streaming algorithm.

It considers all vertices by decreasing degree and *checks* whether it is better to put a vertex at the beginning or at the end of the ordering. More specifically, π is obtained as follows: before placing vertex u, let V_b (resp. V_e) be the vertices that have been placed at the beginning (resp. at the end) of the ordering, and $V_?$ those that are yet to place. The neighbours of u are partitioned in $N_b = N_u \cap V_b$, $N_e = N_u \cap V_e$ and $N_? = N_u \cap V_?$. We consider two options to place u: either just after the vertices in V_b ($\pi_u = |V_b| + 1$), or just before the vertices in V_e ($\pi_u = n - |V_e|$). In either case, u has all vertices of N_b as predecessors, and all vertices of N_e as successors. In the first case, vertices in $N_?$ become successors, which induces a C^{+-} cost $C_b = |N_b| \cdot (|N_e| + |N_?|)$. In the second, the cost is $C_e = (|N_b| + |N_?|) \cdot |N_e|$. The option with the smaller cost is selected. Sorting the vertices by degree requires $\Theta(n)$ steps with bucket sort. Maintaining the sizes of N_b, N_e, $N_?$ for each vertex requires one update for each edge. Therefore, the complexity is $\Theta(m + n)$, or $\Theta(m)$ assuming that $n \in \mathcal{O}(m)$.

Split heuristic

Finally, we propose a heuristic that is faster to achieve but compromises on the quality of the resulting ordering. Degree ordering has been identified as the best solution for mere-listing with algorithm-A++ by Ortmann and Brandes [2013]. We adapt it for C^{+-} by *splitting* vertices alternatively at the beginning and at the end of the ordering π, as shown on Figure 5.10. More precisely, a non-increasing degree ordering δ is computed, then the vertices are split according to their parity: if u has index $\delta_u = 2i + 1$ then $\pi_u = i + 1$; if $\delta_u = 2i$, then $\pi_u = n + 1 - i$. Thus, high degree vertices will have either few predecessors or few successors, which ensures a low C^{+-} cost. On the graph of Figure 5.9, the non-decreasing degree ordering (e, b, g, a, f, d, c) has $C^{+-} = 7$ and the *Split* method leads to (e, g, f, c, d, a, b), which has $C^{+-} = 4$. The complexity of this method is in $\Theta(n)$ like the degree ordering. It is a streaming algorithm based on a centrality mechanism, as described in Section 3.3.3.

While the split procedure can be applied to any ordering, let us see at which condition the resulting cost will be lower. We can consider that the initial ordering π is such

that $\pi_u = u$ for each node u. If V_0 are the even nodes and V_1 the odd nodes, the ordering $Split(\pi)$ will first contain the nodes of V_0 in descending order, then the nodes of V_1 in ascending order. Take a node $u \in V_0$ that has d neighbours including d^- predecessors and d^+ successors; let us assume that they are evenly distributed among V_0 and V_1. After splitting, its $\frac{d}{2}$ neighbours in V_1 become successors; in V_0, its $\frac{d^-}{2}$ predecessors become successors because of the reversion of V_0 in $Split(\pi)$, and its $\frac{d^+}{2}$ successors become predecessors. The contribution of u to $C^{+-}(Split(\pi))$ is thus $(\frac{d}{2} + \frac{d^-}{2}) \cdot \frac{d^+}{2} = \frac{1}{2}d^+d^- + \frac{1}{4}d^+d^+$. Assuming that the degree of a node is not correlated to the parity of its index in π, the same result holds for nodes of V_1. Summing over all the nodes, we get

$$C^{+-}\Big(Split(\pi)\Big) = \frac{1}{2}C^{+-}(\pi) + \frac{1}{4}C^{++}(\pi)$$

The split procedure is useful when the resulting C^{+-} cost is lower than both the initial C^{+-} and C^{++} costs. This is a case when $C^{+-}(Split(\pi)) < \min(C^{+-}(\pi), C^{++}(\pi))$, which is equivalent to $\frac{1}{2}C^{++}(\pi) < C^{+-}(\pi) < \frac{3}{2}C^{++}(\pi)$ given the above equality. In other words, any ordering π for which the cost C^{+-} is similar (between 0.5 and 1.5 times) to the cost C^{++} can be improved using the split procedure. Experimentally, the degree ordering satisfies this double inequality, which explains why splitting it leads to a lower C^{+-} cost.

5.5 Experiments

5.5.1 Experimental setup

Datasets

We use the 12 real-world graphs described in Table 5.4. As this chapter focuses on in-memory triangle listing in large graphs, we selected networks that have between ten million and two billion edges and can therefore be loaded in the RAM of a standard machine. These datasets are standard for evaluating graph algorithms on real-world data: most of them appear in the experiments of Ortmann and Brandes [2013], Danisch et al. [2018] and come from widely-used graph collections [Leskovec and Krevl, 2014, Rossi and Ahmed, 2015], while larger webgraphs are provided by Boldi and Vigna [2004]. All the tested networks are included in the experimental results below. Loops have been removed and the directed graphs have been transformed into undirected graphs by keeping an edge when one existed in either or both directions.

Software and hardware

We release a uniform open-source implementation [3] of A++ and A+− algorithms, as well as the different ordering strategies that we discussed in Section 5.4. The code is in c++ and uses an integer representation that adapts to the size of the graph, allowing for graphs with up 2^{64} edges in principle. It is compiled with gnu make 4 and g++ 8.2 with optimisation flag Ofast and openmp for parallelisation. We run all the programs on an isolated node of a computing cluster to ensure that time measurements are not affected by other processes. The machine is a sgi ub2000 intel xeon e5-4650L @2.6 GHz, 128Gb ram with linux suse 12.3.

[3] https://github.com/lecfab/volt

Dataset (source)	Nodes	Edges	Triangles
skitter ★ (a)	1,696,415	11,095,298	28,769,868
patents ■ (a)	3,774,768	16,518,947	7,515,023
baidu ★ (b)	2,141,301	17,014,946	25,207,196
pokec ▲ (a)	1,632,804	22,301,964	32,557,458
socfba ▲ (b)	3,097,166	23,667,394	55,606,428
LJ ▲ (a)	4,036,538	34,681,189	177,820,130
wiki ★ (a)	2,070,486	42,336,692	145,707,846
orkut ▲ (a)	3,072,627	117,185,083	627,584,181
it ★ (c)	41,291,318	1,027,474,947	48,374,551,054
twitter ▲ (c)	41,652,230	1,202,513,046	34,824,916,864
friendster ▲ (a)	124,836,180	1,806,067,135	4,173,724,142
sk ★ (c)	50,636,151	1,810,063,330	84,907,041,475

Table 5.4: **Datasets used for the experiments**, ranked by number of edges. They represent either web networks ★, social networks ▲ or citation networks ■. The sources are (a) Leskovec and Krevl [2014], (b) Rossi and Ahmed [2015], and (c) Boldi and Vigna [2004].

Our implementation of either triangle listing algorithm can run in parallel because each iteration of the main loop is independent from the others. Among orderings however, only degree and *Split* are easily parallelisable; to be consistent, we use a single thread to compare the different methods. Moreover, the goal of this work is to evaluate the impact of different methods on the speed of triangle listing, which is more straightforward to observe with a single thread.

Regarding the state of the art, the most competitive implementation available for triangle listing is kClist by Danisch et al. [2018], which has already been shown to outperform the previous programs of Makino and Uno [2004] and Latapy [2008]. It uses a recursive algorithm that is equivalent to A+− when $k = 3$. Its data-structure allows to store graphs up to $2^{32} \simeq 4$ billion edges. Besides, Ortmann and Brandes [2013] do not provide the implementation that they used to identify core-A++ and degree-A++ as the fastest methods.

To compare our implementation to the reference implementation of Danisch et al. [2018], we run the following experiment. For each dataset (we limit ourselves to the datasets with under a billion edges in order to obtain indicative results in a short time), we generate various orderings: core, degree, our heuristics, but also depth-first or breadth-first search, random and original orderings, and other variations. For each of them, we then measure the runtime of triangle listing with the program of Danisch et al. and with ours. Out of 136 measures (17 orderings times 8 datasets), our program is faster 131 times, with an average speedup of 1.18, presumably because it does not use recursion.

This result shows that our implementation itself is faster than the existing ones. Therefore, we only use our own implementation of A+− and A++ in the next sections: we exclusively focus on the speedup caused by the vertex ordering, separating it from the speedup originating from the implementation.

5.5.2 Cost and running time are linearly correlated

The aim of this section is to show that the cost functions C^{++} and C^{+-}, which originate from the complexity of A++ and A+−, are accurate estimates of the running time of these algorithms. To do so, we measure the correlation between the running time of mere-

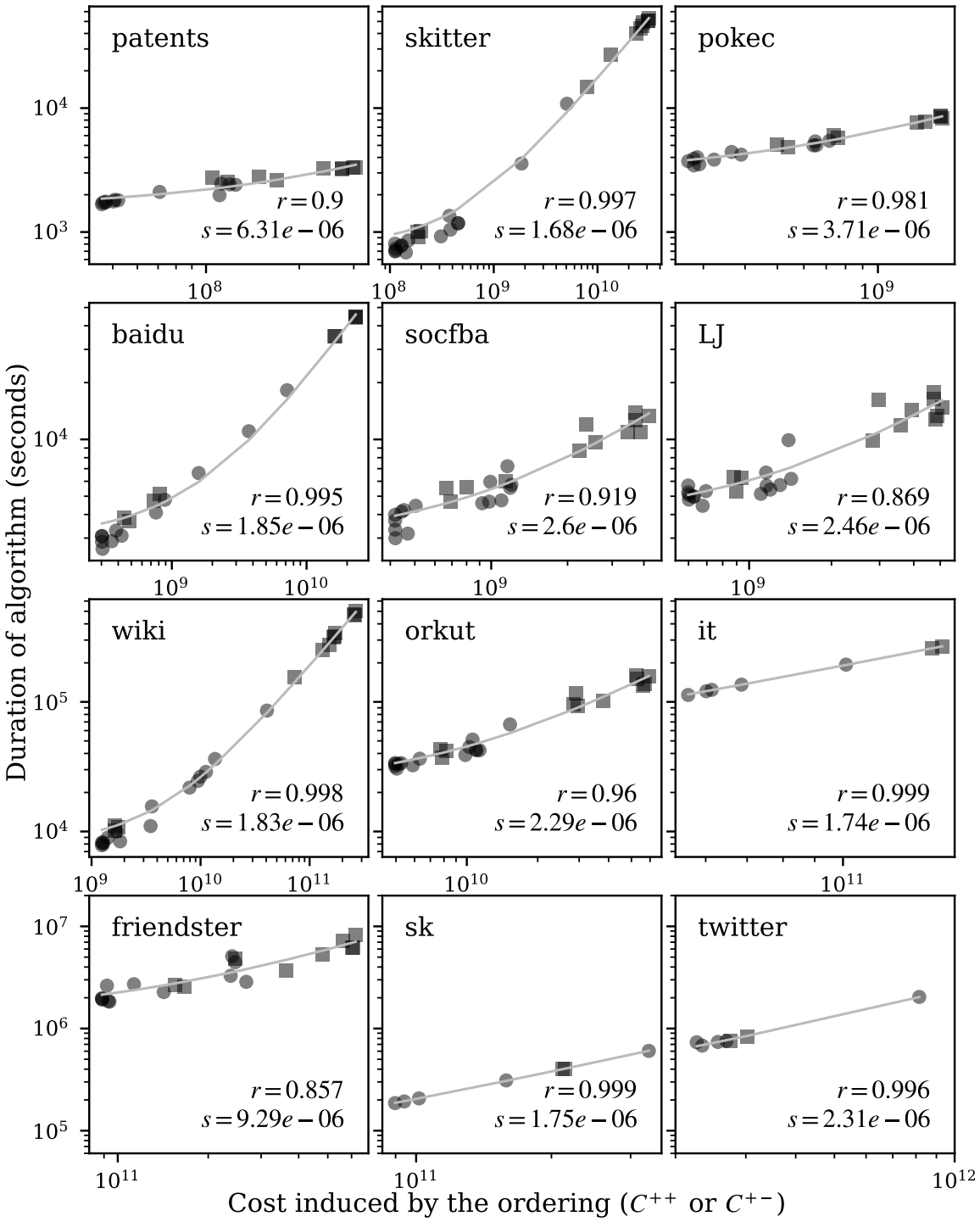


Figure 5.11: **Correlation between execution time and the cost induced by the ordering.** Each mark represents an ordering: circles are for cost C^{+-} and algorithm A+−, squares are for cost C^{++} and algorithm A++ (shades are due to transparency and superposition). Each plot represents a dataset: the line of a linear regression shows the proportionality between cost and time, along with the associated correlation coefficient r and the slope s. Note that the log-scale explains why the lines are curved.

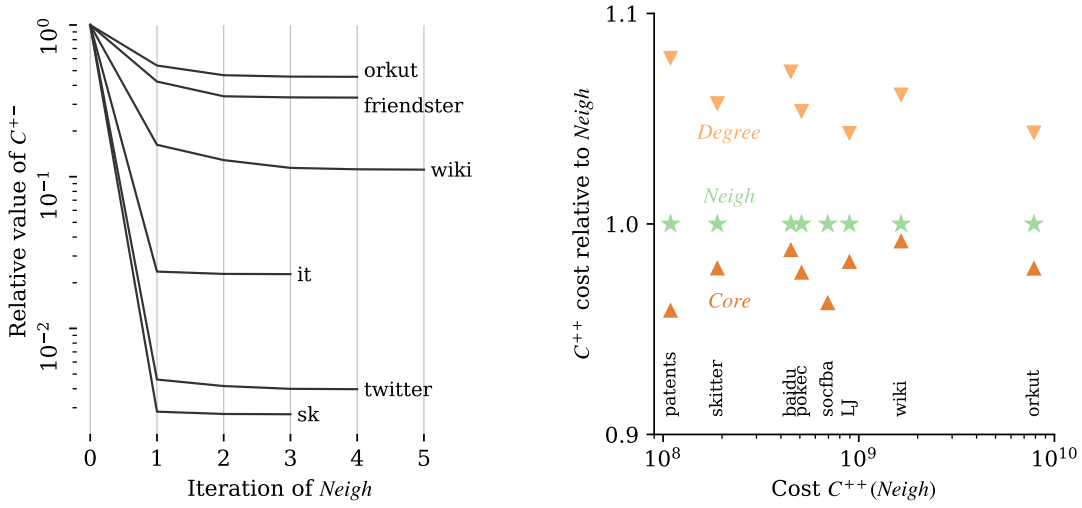
listing with either A++ or A+− and the corresponding cost induced by various orderings (core, degree, our heuristics, but also breadth- and depth-first search, random ordering, etc). In Figure 5.11, we see that the running time for a given dataset correlates almost linearly to the corresponding cost: the lines represent linear regressions. The correlation is above 0.85 on all the datasets, with a median of 0.988. Note that the imperfect correlation of running times across the executions may be caused by the same hardware fluctuations that were investigated in Section 2.3). Note that the slope vary from one dataset to another, the cause of which remains an open question.

Ultimately, the execution time of a listing algorithm is almost a linear function of the cost induced by the ordering, independently of the choice of algorithm: for a given network and ordering π, the execution time of algorithm A++ will be linked to $C^{++}(\pi)$ and the execution time A+− will be linked to $C^{+-}(\pi)$. For this reason, reducing the costs actually improves the running time, as we will see in the next experiments.

5.5.3 Assessing the *Neigh* heuristic

Among our three heuristics to lower the C^{+-} cost, *Neigh* is the one where we expect the best results in terms of C^{+-}, while the others focus on a short ordering time. For this reason, we need to assess the efficiency and stability of *Neigh*.

Convergence. The *Neigh* heuristic consists in I iterations of a loop over all the nodes of the graph. The process stops when a given improvement threshold is reached, but there is no guarantee that this will happen in reasonable time. In practice, Figure 5.12a shows that even for the biggest datasets, the threshold is reached in a few iterations.



(a) **Convergence of *Neigh* heuristic for cost C^{+-} on the biggest datasets.** Each line shows the relative reduction of C^{+-} after a certain number of iterations. They stop when relative improvement is under 10^{-2}.

(b) **Value of *Neigh* heuristic applied on C^{++} cost function, compared to degree and core orderings.** Values are normalised by $C^{++}(Neigh)$. Observe that *Neigh* is always between core and degree.

Figure 5.12: **Assessing the *Neigh* heuristic.** Convergence happens after a few iterations and reaches a C^{++} cost that has a comparable quality with degree or core orderings.

Stability. *Neigh* is based on the search of a local minimum, which depends on the initial ordering. Mathematically, it is possible that some initial orderings lead to a very low cost while some other lead to a poor local minimum. To show that this is not the case in practice, we take the *orkut* dataset and apply the heuristic on 100 distinct random orderings. The best cost obtained for C^{+-} is $42.1 \times m$. The worst result is only 2.9% higher, and the relative variation over 100 iterations is 0.7%. This result means that the method is stable and that many local optima have close C^{+-} costs.

Test for C^{++}. Our final objective with the *Neigh* heuristic is to reduce the C^{+-} cost. As there is no benchmark for this cost, we propose to test the relevance of the *Neigh* optimisation procedure on the objective function C^{++}, for which core and degree orderings can be taken as benchmarks. We run the *Neigh* heuristic with the objective function C^{++} and compare the resulting ordering π to degree and core orderings, δ and γ. In Figure 5.12b, we observe that $C^{++}(\pi)$ is on average only 2.3% higher than $C^{++}(\gamma)$, and 7% lower than $C^{++}(\delta)$. To give an idea of the improvement compared to a random node ordering, $C^{++}(\pi)$ is also 17 times lower than $C^{++}(rand)$.

This comparison highlights two things: first, although previous works did not attempt to reduce C^{++} explicitly, they selected the core and degree orderings for A++ because their C^{++} costs are much lower than random. Second, our *Neigh* heuristic reaches comparable values, which indicates that its principles work well: we can thus be confident that the same heuristic applied on C^{+-} will reduce the number of operations of A+−.

5.5.4 The new orderings outperform previous listing methods

We compare our methods (*Neigh*, *Check* and *Split* with A+−) to the state of the art for mere-listing (core-A+− for Danisch et al. [2018] and core-A++ for Ortmann and Brandes [2013]) and for full-listing (degree-A++ for Ortmann and Brandes). Recall that we only use our implementation to ensure that the results describe the influence of node orderings, and not the implementation techniques.

Figure 5.13 gives an overview of the performance of all the methods. The top charts present the running times of the three state-of-the-art methods for all datasets. The bottom charts show the speedup of our methods compared to the fastest existing one: we take for each dataset the fastest of the three existing methods as a reference, and the speedup is defined as the duration ratio between this reference and our method. In each case, we distinguish the mere-listing task (left) from the full-listing task (right). Exact runtimes of the best existing and of the methods proposed in this chapter are reported in Table 5.5.

Neigh outperforms previous mere-listing methods

Let us first consider the mere-listing task. Regarding existing methods, we can see at the top left of Figure 5.13 that there is no clear winner for mere-listing: the durations of both A++ methods are very close, but core-A+− can be between 1.4 times faster and 2.4 times slower depending on the dataset. This may explain why Ortmann and Brandes [2013] and Danisch et al. [2018] did not agree on the fastest method.

For our methods, the main result is that *Neigh*-A+− is always faster than the best previous method. Looking at the bottom left of Figure 5.13, we see that the speedup is 1.38 on average and ranges from only 1.02 on *twitter* to 1.71 on the *it* dataset. *Check*-A+− performs almost as well, with a 1.32 average speedup ranging from 1.10 to 1.60; it is even faster than *Neigh*-A+− on two of the datasets. *Split*-A+− is a little slower, which is expected because this ordering is designed to be obtained quickly and does not reduce C^{+-} as efficiently as

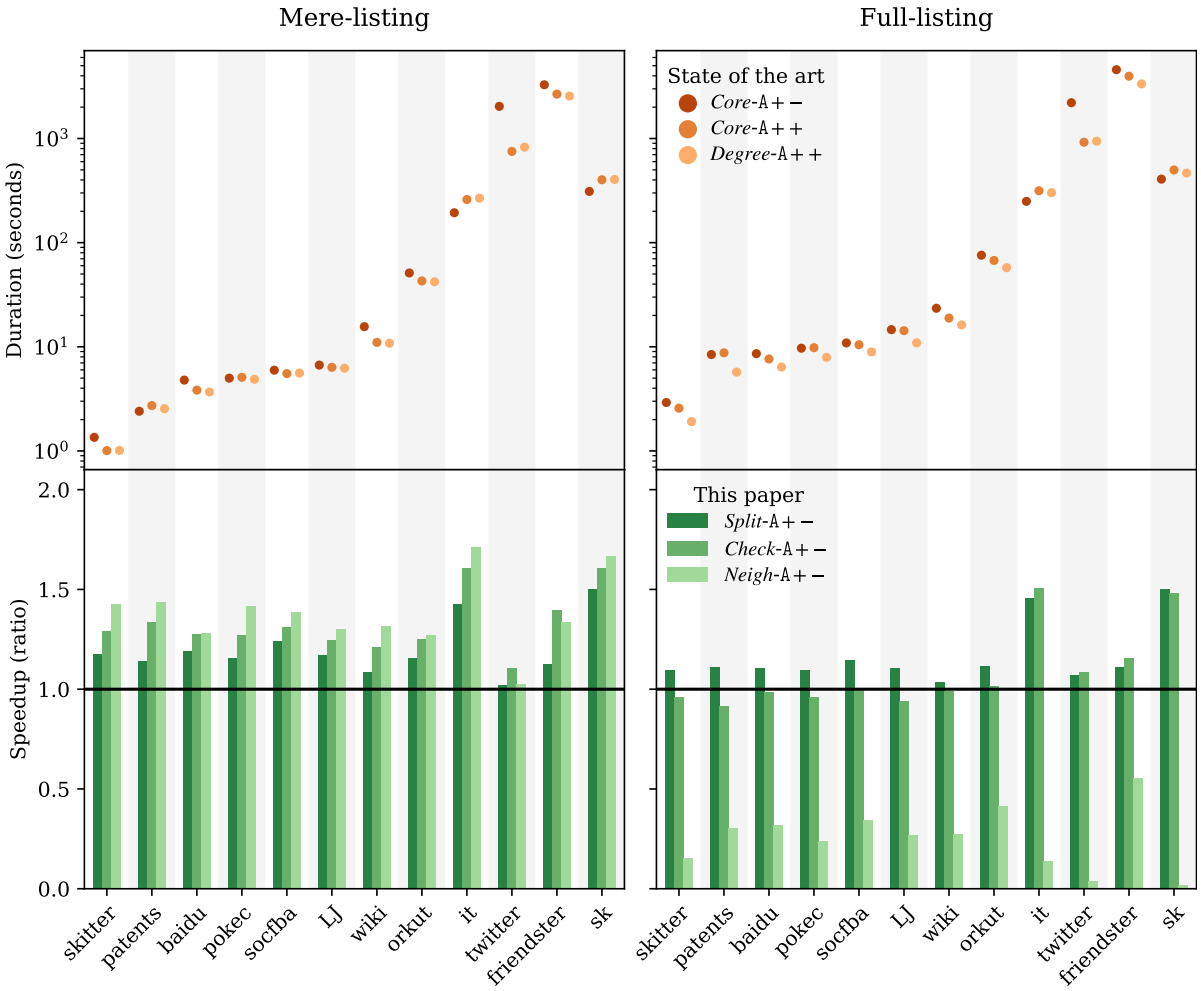


Figure 5.13: **Comparison of state-of-the-art methods and speedup of our methods.** The top charts show the runtime of the three state-of-the-art methods; depending on the dataset, the fastest method is not always the same. The bottom charts show the speedup of our three methods against the fastest existing method of each dataset. On the left, for mere-listing, we see that our three heuristics consistently outperform the three state-of-the-art methods, and that *Neigh* or *Check* are the fastest. On the right, for full-listing, *Neigh* is not efficient but *Split* is always faster than existing methods and *Check* is faster on bigger datasets.

	mere-listing		full-listing	
dataset	state of the art	this work	state of the art	this work
skitter	1.00s	0.71s	1.91s	1.75s
patents	2.40s	1.67s	5.71s	5.15s
baidu	3.68s	2.87s	6.38s	5.77s
pokec	4.87s	3.44s	7.91s	7.21s
socfba	5.52s	3.98s	8.92s	7.79s
LJ	6.23s	4.79s	10.91s	9.88s
wiki	10.82s	8.22s	16.23s	15.65s
orkut	42.11s	33.09s	57.47s	51.60s
it	3m13	1m53	4m09	2m45
twitter	12m31	11m20	15m21	14m08
friendster	42m36	30m31	55m47	48m13
sk	5m10	3m06	6m47	4m31

Table 5.5: **Duration of triangle listing of existing methods against our methods.** For each dataset, we compare the fastest state-of-the-art method against the fastest of our methods. Recall that mere-listing only takes into account the runtime of the listing algorithm ($\mathrm{A}{+}{+}$ or $\mathrm{A}{+}{-}$) while full-listing also counts the graph loading time and the ordering time.

our other heuristics. However it still consistently outperforms all the previous methods, with a 1.20 average speedup. The conclusion is that our heuristics *Neigh*, *Check* and *Split* manage to produce orderings that induce significantly lower C^{+-} costs, which translates directly into short running times for mere-listing with $\mathrm{A}{+}{-}$.

Split outperforms previous full-listing methods

For full-listing, the top right chart of Figure 5.13 shows that degree-$\mathrm{A}{+}{+}$ is the fastest for nine of the twelve datasets. This result is consistent with the result reported by Ortmann and Brandes [2013], who specifically addresses full-listing. Yet for bigger datasets, which were not studied in previous works, the result is not as clear: core-$\mathrm{A}{+}{+}$ yields close running times to degree-$\mathrm{A}{+}{+}$, and core-$\mathrm{A}{+}{-}$ outperforms both on two instances.

As for our methods, the main result is that *Split*-$\mathrm{A}{+}{-}$ is always faster than previous methods. The speedup compared to existing methods is 1.16 on average, and it ranges from 1.04 on *wiki* to 1.50 on *sk* dataset. *Check* also gives very good results: on medium datasets, it is a bit slower than degree-$\mathrm{A}{+}{+}$, but it outperforms all state-of-the-art methods on large datasets (*it, twitter, friendster, sk*), and it even beats *Split* on three of them. This hints at a transition effect: the *Check* ordering has a lower C^{+-} value but it takes $\Theta(m)$ steps to compute, while *Split* only needs $\Theta(n)$; for larger datasets, the listing step prevails, so the extra time spent to compute *Check* becomes profitable. Note that the *Neigh* heuristic is not competitive here (speedup under one) since it has a long ordering time compared to other methods.

5.6 Conclusion

In this chapter, we address the issue of in-memory triangle listing in large graphs. We formulate explicitly the computational costs of the most efficient existing algorithms, and investigate how to order vertices to minimise these costs. After proving that the optimisation problems are NP-hard, we propose scalable heuristics that are specifically tailored to reduce the costs induced by the orderings. We show experimentally that these methods outperform the current state of the art for both the mere-listing and the full-listing tasks. The *Check* heuristic offers a good compromise for either task, and we set it as default in our open-source implementation.

Preliminary investigations indicate that the mere-listing step takes more importance as graphs grow larger, which hints that our listing methods would be all the more efficient for future, larger datasets. Moreover, full-listing includes the time spent to load the graph and to compute the ordering; these two tasks can hardly be accelerated, as the complexity of *Split* is linear in the number of nodes of the graph. For this reason, we think that improving triangle listing will require to find orderings that are fast to compute and that further accelerate the mere-listing step. Modifications of the standard algorithms using more elaborate data-structures to compute list searches and intersections may also come into play, which would involve new complexity costs and new orderings.

The method of this chapter could in principle extend to other problems. We give below our insights into some of them and we detail the issues that we faced. One natural extension is to use similar vertex ordering heuristics in the more general case of clique listing. This has been done by Danisch et al. [2018] using core ordering, and refined by Li et al. [2020] using a pruning technique involving graph colouring. These methods use broad bounds of complexity and could be improved with a tighter study of the cost of clique listing algorithms. Yet, formulating appropriate cost functions like C^{++} and C^{+-} is not straightforward for cliques: while the existence of a triangle is checked from a path of length two, the existence of a k-clique derives from a $(k-1)$-clique with an out-going edge, and this clique itself depends on a $(k-2)$-clique etc. The costs that we obtain are too specific to run an efficient optimisation procedure, and they involve to rank nodes not by their degree or degeneracy, but by the number of times they participate in a precise type of pattern; this count is costly to find, which hinders any acceleration to the overall clique listing algorithm.

Beyond cliques, we also considered the listing of general k-motifs, which consists in finding all the instances of a given connected subgraph of k nodes. Two issues rule out the methods that we applied to triangles. First, the technique to avoid redundancy by orienting the edges does not work for arbitrary motifs: consider $k = 3$ and an open wedge motif (edges u, v and v, w forming a $\vee$ shape). As in the case of triangles, each wedge could be counted twice: (u, v, w) and (w, v, u). If edges are oriented, however, some wedges are not counted at all. To avoid missing motifs, either the algorithm has to take all the possible orientations into account, or the orientation of the edges has to be more constrained. Second, the number of possible motifs increases dramatically with k. Pinar et al. [2017] break motifs into smaller ones to mitigate this increase, which requires to specialise the listing algorithm even further.

For the related problem of triangle sampling, Turk and Turkoglu [2019] use the core ordering because it reduces the number of low-hinge wedges. It is the equivalent of using algorithms with cost C^{++} for triangle listing, and we could suggest to use C^{+-} with *Split* ordering instead. But their wedge sampling procedure relies on checking each pair of out-neighbours: the corresponding cost is more precisely $\sum_{u \in V} \binom{d_u^+}{2} = \frac{C^{++}}{2} - m$. As hinted

by the expected values of these costs for random graphs in Proposition 5, this new cost is closer to C^{+-} on random graphs. There is therefore little to gain by using the methods of this chapter.

Finally, our contribution may improve the work of Pashanasangi and Seshadhri [2021] on triangle counting in temporal graph. They use degeneracy ordering to list the static triangles, after which they include the temporal information of edges to count the occurrences of the corresponding dynamic triangles within a predetermined time window. The dynamic part is independent of the choice of ordering and thus cannot be impacted by our methods. The static enumeration, on the other hand, could be accelerated by using *Neigh*, *Check* or *Split* ordering and a variation of algorithm A$+-$. Our efforts in this direction show that the time spent for the static enumeration is negligible compared to the time for dynamic counting, which means that the gain would be positive, but insignificant.

In summary, our work shows a situation where studying the precise cost of an algorithm instead of its asymptotic complexity allows us to design specific methods to accelerate its execution. A variety of algorithms may benefit from this approach even though the result does not extend automatically to other mining problems. In the case of triangle listing, the node ordering operates the link between the mathematical cost and the running time, which leads to a significant speedup when using tailored orderings.

Chapter 6

Orderings for quality certification

Certifying the quality of vertex cover heuristics

Summary

While there is a demand for scalable algorithms with the ever-increasing access to large datasets, some algorithmic problems are hard to solve on large real-world instances. Part of them, such as the vertex cover problem, have a k-approximation algorithm: the result is guaranteed to be within a factor k of the optimum one. But even then, real-world applications obtain better results with fast algorithms that have no theoretical guarantees, called heuristics. There is however no indication on how far these heuristic results are from optimum.

To address this issue, we propose a method to certify the quality of a heuristic on a given instance. The quality certification consists in comparing the experimental result to a bound of the optimum value, obtained through a different heuristic. We show two ways of obtaining bounds for the vertex cover problem: one with a bound that is specific to that problem, and one using a k-approximation algorithm, that may generalise to other problems.

We test our approach on 114 real-world networks with up to three billion edges. The experiments show that the certified quality is within 1.11 for all tested instances and 1.01 for 78 of them. It means that there is a proof that the vertex covers found with heuristics are at most 1.11 times larger than the minimum one. In conclusion, our method provides valuable quality certificates for existing heuristics on specific instances, without loosing on scalability. As it generalises to algorithmic problems with a k-approximation, it opens a door for further research and for deployment in real-world applications.

Contributions

* Propose a quality certification framework using heuristics and lower-bounds.
* Adapt it to the vertex problem using existing heuristics with two types of lower-bounds.
* Present scalable greedy algorithms to address both aspects of the certification.
* Give empirical evidence that the certified quality is good on real-world datasets.
* Release an efficient open-source implementation of all considered methods.
  https://github.com/lecfab/certifVC

Publications & talks

* *Vertex cover quality certification on real-world networks*
  Lécuyer, Tabourier, and Magnien [2023b], submitted ESA.
* Presented at JGA'22, FRCCS'23.
* Seminar in Milan.

6.1 Introduction

The curse of applied algorithmics is that many of the most crucial problems are NP-hard, which generally means that exact algorithms cannot scale to massive datasets. Some exceptions exist, like the problem of enumerating maximal cliques for which Eppstein et al. [2013] show an exact algorithm that handles real-world networks with million of nodes despite its exponential complexity. Yet, in general, there is a split between exact methods that push the scalability further and further, and quick methods called heuristics that get closer and closer to optimum results. The latter is particularly efficient on real-world instances such as social networks, web graphs or biological networks, which rarely resemble worst-case scenarios and allow heuristics to give excellent results. However, while heuristic results can be compared to one another, it is not possible to measure how far they are from optimal.

This work introduces a method to certify the quality of a result on a specific instance. It combines a heuristic for the initial problem with another heuristic that gives an instance-specific bound of the optimal solution. Together, the approximate result and the bound allow us to compute a practical approximation ratio for a given instance. We call this approach a *quality certification*. We showcase it on the the minimum vertex cover problem, a famous NP-complete graph problem that has applications in network robustness [Sáenz-de Cabezón and Wynn, 2014], wireless communication [Yigit et al., 2021], virus transmission [Sander et al., 2008] and image rendering [Boros and Gurvich, 2007].

When addressing a hard problem such as vertex cover, one faces a trade-off between speed and guarantee of quality. On the one hand, fast heuristics such as the ones of Cai et al. [2017] obtain results that are close to the optimum solution, as confirmed by Gomes et al. [2006] when the optimum is known. Yet, in the more interesting case when the optimum is unknown, it is not possible to measure how accurate the heuristic result is. On the other hand, exponential algorithms for exact solutions can be extremely fast on real-world networks: the 2019 PACE challenge [Dzulfikar et al., 2019] fostered efforts towards quick and exact algorithms for vertex cover, and the laureate implementation by Hespe et al. [2020] solves some graphs of millions of nodes in a few seconds. Still, our experiments show that it fails to solve the problem in reasonable time for larger or more complex graphs.

A quality certification helps with this trade-off: it takes a dataset and gives both an approximate result and a certificate of its quality, defined as the ratio between the heuristic result and a bound on the optimum value. For example, the shortest path between two cities is lower-bounded by the distance as the crow flies; the certified quality of a path is then given by the ratio between its length and the lower-bound. Such bounds are hard to obtain in general. The key insight of this chapter is that multiple problems of interest can be bounded empirically, using heuristics to obtain a high lower-bound – or a low upper-bound – on a specific instance. We propose two practical bounds for the minimum vertex cover and we show that they enable us to give precise quality certificates.

The rest of the paper is organised as follows. Section 6.2 introduces the definitions and notations used in the study. Section 6.3 reviews related works on exact and approximate solutions for the vertex cover problem, as well as on notions related to certification. Section 6.4 describes the proposed quality certification approach and its specifics in the case of the vertex cover problem. Finally, Section 6.5 presents experimental results that demonstrate the relevance and scalability of the method on more than a hundred real-world networks.

6.2 Background and notations

We consider an undirected unweighted simple graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges. The set of neighbours of a vertex u is denoted $\mathcal{N}_u = \{v \,|\, \{u, v\} \in E\}$, and its degree is $d_u = |\mathcal{N}_u|$. The edges incident to a subset $W \subseteq V$ are all the edges with at least one extremity in W. A solution is *optimal* when it cannot be modified into a better solution by adding or removing an element, and *optimum* if it is as good as any other solution.

We now present the definition of the vertex cover problem, at the core of this work.

Definition 9 (Vertex cover, minimal, minimum) *A vertex cover is a set of nodes C that is incident to every edge of the graph: $\forall \{u, v\} \in E : u \in C$ or $v \in C$. To simplify, it can be referred to as a* cover. *A vertex cover C is* minimal *if removing any node uncovers an edge: $\forall v \in C, C \backslash \{v\}$ is not a cover. It is* minimum *if it is as small as any other vertex cover.*

Knowing if there exists a vertex cover of a given size is NP-complete, thus finding a minimum vertex cover is NP-hard, as well as the complementary problem of finding a maximum independent set. Both problems figure in the 21 NP-complete problems described in 1972 by Karp [1972] [1]. They are complementary to each other, since the nodes that are not in a vertex cover form an independent set. The minimum vertex cover has a 2-approximation algorithm based on a maximal matching:

Definition 10 (Matching, maximal, maximum, minimum maximal) *A matching M is a set of independent edges: $\forall e, f \in M,\ e \cap f = \emptyset$. A matching M is* maximal *when all the edges of the graph are incident to its nodes: $\forall e \in E, \exists f \in M, e \cap f \neq \emptyset$. A* maximum *matching is a maximal matching that has the largest possible size. A* minimum maximal *matching is a maximal matching that is as small as any other matching.*

Definition 11 (Cover of a matching) *The nodes of any maximal matching M form a vertex cover noted C_M and called the cover of the matching:*

$$C_M = \bigcup_{\{u,v\} \in M} \{u\} \cup \{v\} \subseteq V.$$

Property 11 (Cover approximation) *For any maximal matching M, the vertex cover C_M is at most twice as large as any cover. For a minimum cover C^* in particular, $\frac{1}{2}|C_M| \leq |C^*| \leq |C_M|$.*

Proof: The edges of M are independent. For each of them, C_M contains 2 nodes, but C^* must contain at least 1 node in order to cover this edge.

□

Let us also define the concept of clique cover, that we later use to obtain lower-bounds for the minimum vertex cover:

Definition 12 (Clique cover, minimal, minimum) *A clique cover of size p is a partition of the nodes into cliques $\mathcal{P} = K_1, \ldots, K_x$: $K_1 \cup \cdots \cup K_x = V$, if $i \neq j$ then $K_i \cap K_j = \emptyset$, and $\forall u, v \in K_i,\ \{u, v\} \in E$. The clique cover $\mathcal{P}$ is* minimal *if no pair of its elements form a larger clique together: $\forall K, K' \in \mathcal{K}, \exists u \in K, v \in K', (u, v) \notin E$. It is* minimum *if it is as small as any clique cover.*

[1] The list contains Node Cover (another name for Vertex Cover), and Set Packing which is equivalent to Independent Set and shares its approximation properties.

Algorithm 6 – Edge-greedy 2-approximation for vertex cover

Input: priority function $\eta : E \to Z$
1: start with empty cover: $C_M \leftarrow \emptyset$
2: **while** there are uncovered edges **do**
3: take uncovered edge $\{u, v\}$ with highest priority $\eta(\{u, v\})$
4: add u and v to cover: $C_M \leftarrow C_M \cup \{u\} \cup \{v\}$
5: **return** C_M

6.3 Related Work

6.3.1 Solutions, approximations and heuristics for vertex cover

Hardness of the vertex cover problem

On top of the hardness results for vertex cover and independent set, let us see some other approximability results. For vertex cover, the approximation ratio due to maximal matchings can be lowered to $2 - o(1)$ with the more sophisticated methods of Karakostas [2005], but it is thought that no lower constant ratio can be achieved [Khot and Regev, 2008, Bazzi et al., 2018] and proven impossible to fall under 1.36 unless P=NP [Dinur and Safra, 2005]. As for the independent set problem, it is hard to approximate within any constant factor on general graphs; on line graphs, it is equivalent to maximum matching and therefore has a polynomial time algorithm.

Tighter approximation ratios exist for graphs that have typical properties of real-world networks, such as bounded degrees [Halldórsson and Radhakrishnan, 1997, Avis and Imamura, 2007], high clustering [Bläsius et al., 2020], or a power-law degree distribution [Gast and Hauptmann, 2014]. Reduction and kernelisation rules, surveyed by Fellows et al. [2018], diminish the instance size and can lead to better approximation guarantees [Asgeirsson and Stein, 2007] or faster executions [Hespe et al., 2020] on specific instances.

The two problems have been generalised to weighted graphs [Cai et al., 2018, Xiao et al., 2021], dynamic networks [Akrida, 2020, Assadi et al., 2018] or partial coverage [Hochbaum, 1998], and analysed in the quadratic programming [Pandey and Punnen, 2018] and massively parallel settings [Ghaffari et al., 2018].

2-approximation by matchings

To obtain a small vertex cover, it is possible to use matchings. Indeed, we have seen in Section 6.2 that the cover of a matching provides a 2-approximation for vertex cover, which is the best known constant-factor approximation on general instances. It is NP-hard to find a maximal matching of minimum size, but a maximum matching can be found in polynomial time, for instance with the blossom algorithm of Edmonds [1965].

Algorithm 6 builds a vertex cover by repeatedly adding both extremities of an uncovered edge to the cover until all edges are covered. In this algorithm, we use a function η that yields an ordering of the edges of the graph, and call it a *priority function*. We will choose priority functions that makes the complexity of this heuristic linear. As all edges are covered at the end of the execution, the algorithm creates the cover of a matching and thus provides a 2-approximation of the minimum vertex cover. The left and middle examples of Figure 6.1 illustrate two such maximal matchings obtained with different priority functions on the same toy graph.

Algorithm 7 – Node-greedy heuristic for vertex cover

Input: priority function $\nu : V \to Z$
1: start with empty cover: $C \leftarrow \emptyset$
2: **while** there are uncovered edges **do**
3: take node u with highest priority $\nu(u)$
4: add u to cover: $C \leftarrow C \cup \{u\}$
5: **return** C

Heuristics without constant-factor approximation guarantee

While a matching can be used to find a vertex cover of small size, other heuristics without constant-factor approximation guarantees sometimes result in smaller covers in practice. A lot of attention has been given to the design of such heuristics, among which *FastVC* the one of Cai et al. [2017] is considered as the state of the art. A well-known example, that we call node-greedy, is described in Algorithm 7. It has a similar structure to Algorithm 6, but only one node is added at each iteration instead of two. Therefore, the associated priority function ν is defined over the nodes. A standard *high-degree first* priority translates as $\nu_{high}(u) = x_u$, with x_u the number of neighbours of u that are not in the cover at a given step. In that case, the heuristic provides an approximation ratio in $\mathcal{O}(\log(\Delta))$, where Δ is the maximum degree of a node in the graph [Avis and Imamura, 2007]. We illustrate this strategy on the right example of Figure 6.1.

In terms of complexity, the main loop of Algorithm 7 achieves at most n iterations, and updating the priority function ν depends on its definition. ν_{high} priority function needs $\mathcal{O}(m)$ updates, and each of those can be done in $\Theta(1)$ with a bucket implementation, which leads to an overall complexity in $\mathcal{O}(n + m)$.

To ensure that the cover is minimal, a post-processing removes the nodes that have all their neighbours in the cover, without affecting the asymptotic complexity. Such greedy algorithms are known to be scalable and to give small covers [Gomes et al., 2006, Angel et al., 2012], and they become highly accurate when extra reduction rules and local search heuristics are added [Cai et al., 2017].

6.3.2 Quality certification

Suppose that a cover has been obtained using a given heuristic, then only theoretical guarantees, if they exist, give indication about how far this cover is from optimality. In Section 6.4, we will detail our approach that consists in combining a heuristic cover with lower bounds of the minimum value. Together, the two values lead to an experimental approximation factor which is consistently much lower than the theoretical guarantees.

Similar approaches have been successfully applied on a variety of algorithmic problems. A method has been proposed by Dolev and Sadetsky [2009] for algorithms that have a k-approximation or an approximation scheme, and supported with some experiments for the problems of knapsack, max 3-sat, and maximum bounded 3-dimensional matching. They define a heuristic certificate as the ratio between the value of a heuristic result and a lower-bound on the minimum value for a given instance. However, the paper presents limited evidence of the efficiency of the method. Our work extends the method to address the vertex cover problem and highlights its relevance with thorough experiments.

The problem of finding the diameter of a graph, defined as the maximum distance between two nodes, can be solved exactly in polynomial time by computing the shortest

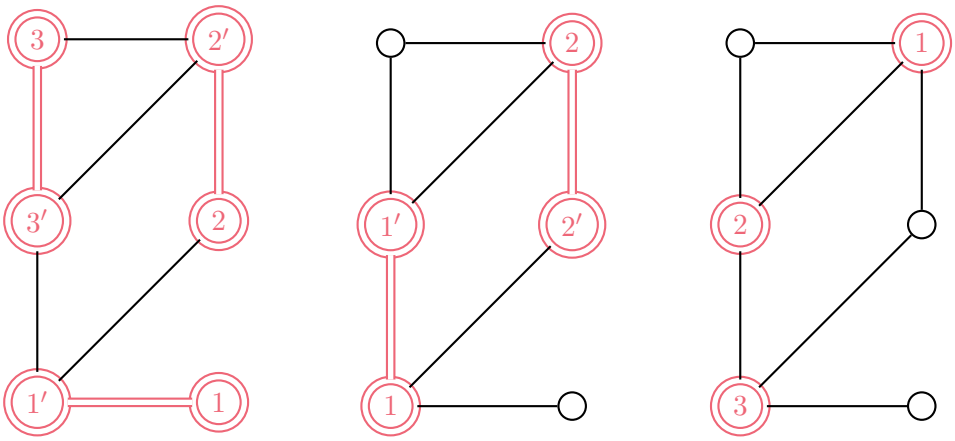


Figure 6.1: **Greedy heuristics for vertex cover.** On a toy example, we show the vertex covers obtained from different greedy heuristics. In all cases, the numbers indicate the step at which a node has been selected by the corresponding algorithm. On the left and middle cases, the red double lines indicate the matching that produced the cover. Left: edge-greedy (Algorithm 6) with low-degree first η_{low} leads to a maximum matching. Middle: edge-greedy with high-degree first η_{high} gives a matching with 4 nodes. Right: node-greedy (Algorithm 7) with high-degree first ν_{high} gives a cover of 3 nodes; it is optimum because the left cover has 6 nodes and is a 2-approximation.

path between any pair of nodes. Yet, the running time of such a procedure is deterrent in practice on large networks. Several upper- and lower-bounds have been devised to obtain a fast guarantee on the diameter Magnien et al. [2009]. On the one hand, the eccentricity of any node, defined as the longest distance found by running a breadth-first search from this node, gives a lower-bound on the diameter; various heuristics exist as to how to select a node of high eccentricity and obtain a large lower-bound empirically. On the other hand, the diameter of the graph is lower than the diameter of a subgraph obtained by removing edges, for instance in a spanning tree. These bounds have been adapted for exact algorithms to only explore the paths that can make them tighter, leading to algorithms that scale linearly in practice [Takes and Kosters, 2011, Crescenzi et al., 2013]. A definition of certificate has been proposed for this problem by Dragan et al. [2018]: to prove that the diameter of a graph is a value D, a certificate is a pair made of a node of eccentricity D used as a lower-bound, and a set of nodes that acts as an upper-bound by proving that all other nodes have eccentricity at most D.

Other works mention bounds to the optimisation problems that they address. For instance, Rossi and Ahmed [2014] test various heuristics for the graph colouring problem empirically, and use a greedy algorithm to find a large clique, which acts as a lower-bound on the minimum number of colours. For the densest subgraph problem, the quadratic optimisation algorithm of Danisch et al. [2017] converges towards an exact solution and indicates an upper-bound on the maximum density.

6.4 Quality certification of the vertex cover problem

This section develops the general method of quality certification and shows how to apply it to the minimum vertex cover problem.

6.4.1 Principle and formulation for the vertex cover problem

The key insight of this work is that coupling the heuristic for a hard problem with another heuristic that gives a complementary bound on the optimum value can result in strong guarantees on a specific instance. The quality certification is then defined as the ratio between an approximate value and the bound on the optimum value. Let us now formulate this principle more formally for the vertex cover problem.

Consider a graph whose unknown minimum value for vertex cover is c^*. As we have seen, heuristics have been designed to obtain an approximate value $s \geq c^*$, with the goal of being as close to c^* as possible; we call such a heuristic a *solution-heuristic*. Now suppose that another algorithm, that we call *lower-bound-heuristic*, produces a positive lower-bound $b \leq c^*$ for this problem. It means that the unknown value c^* satisfies $c^* \in [b, s]$. This implies that $c^* \leq s \leq \frac{s}{b}c^*$. So, by definition, the value s is a $\frac{s}{b}$-approximation of the optimal value c^* for this instance. A natural goal is then to reduce the interval $[b, s]$ by finding a solution-heuristic with a low value s and a lower-bound-heuristic with a high value b.

6.4.2 Certification of the vertex cover problem with a 2-approximation

In this section, we use the counter-intuitive idea that an appropriate lower-bound-heuristic can be found by designing a *poor approximation*: if a solution-heuristic is proven to be a k-approximation, its result s satisfies $s \leq k \cdot c^*$, which also provides a lower bound $b = \frac{s}{k}$. While solution-heuristics are usually designed to output low values s, we propose to adapt them and favour high values, which translate into tighter lower-bounds.

The poor approximation strategy can be used to turn the 2-approximation algorithm for vertex cover into both a solution-heuristic and a lower-bound-heuristic. As discussed in Section 6.3, a maximal matching provides a 2-approximation C_M of the vertex cover problem, which means that Algorithm 6 can be used as a lower-bound-heuristic: its output C_M always satisfies $\frac{1}{2}|C_M| \leq |C^*|$. Following this idea, the poor approximation strategy consists in making C_M as large as possible: this increases the lower-bound, and hence certifies a better quality ratio. Note that this strategy can be related to a primal-dual optimisation, as maximum matching is the dual linear program of vertex cover. We define the notion of vertex cover certificate by matching:

Definition 13 (Vertex cover certificate by matching) *Given a graph, a vertex cover certificate by matching is a couple (C_M^+, C^-), where C_M^+ is the cover of a matching vertex containing $2b$ nodes, and C^- is a vertex cover of s nodes. C^- is certified to be within a factor $\frac{s}{b}$ of the minimum cover size and the ratio $\frac{s}{b}$ is called the certified quality by matching.*

Priority functions for lower-bound-heuristics

We should look for large matchings associated to large covers in order to obtain high lower-bounds. Selecting nodes with low degree is likely to result in a large cover C_M^+, as more edges can intuitively be packed if their nodes have small degree. Thus, we define the low-degree first priority function $\eta_{low}(\{u, v\}) = -\min(x_u, x_v)$, where x_u is the number of neighbours of node u that are not in the cover at the current step.

Ties are broken with the x value of the other node, namely $-\max(x_u, x_v)$. We illustrate this strategy on the left example of Figure 6.1.

Instead of a maximal matching obtained with a greedy heuristic, it is also possible to look for an exact maximum matching. The maximum matching problem can be solved

with the blossom algorithm of Edmonds [1965] in polynomial time $\mathcal{O}(n^2m)$. Other algorithms have a lower complexity in $\mathcal{O}(\sqrt{n}m)$ [Micali and Vazirani, 1980, Blum, 1990] and a linear-time approximation scheme exists [Duan and Pettie, 2014].

Defining solution-heuristics

It is possible to use Algorithm 6 as a solution-heuristic aiming for a small cover. The intuition is that a small cover C_M^- may be obtained by selecting nodes with high degree, because they will cover more edges. With x_u defined similarly as above, the high-degree first priority function is $\eta_{high}(\{u, v\}) = \max(x_u, x_v)$. This is the method illustrated in the middle example of Figure 6.1. However, we observe in practice that this method is not as efficient as other heuristics.

Indeed, any heuristic producing a small cover can be used as a solution-heuristic even if it is not a 2-approximation, and in practice Algorithm 6 outputs larger covers than the node-greedy method of Algorithm 7 with ν_{high} priority function. These heuristics are linear. We will see that the FastVC heuristic [Cai et al., 2018], as expected, performs best, at the cost of a higher computation time.

6.4.3 Improving the certification for vertex cover

In some cases, we can expect that the certified quality by matching will not improve on the theoretical guarantee. Indeed, consider the case of a clique of size k: the matching-based lower-bound-heuristic provides a maximum matching containing all k nodes of the graph, thus the lower-bound is $\frac{k}{2}$ (we consider that k is even for simplicity). Although the minimum vertex cover has $k - 1$ nodes, the certified quality by matchings is $\frac{2(k-1)}{k}$, which tends to 2 for high values of k.

Based on this observation, we propose another bounding strategy that overcomes this issue. In any clique of k nodes, a vertex cover requires at least $k - 1$ nodes. Thus, if we partition the nodes of the graph into p cliques of sizes $k_1, \ldots, k_p$, the vertex cover requires at least $\sum_{i=1}^{p} k_i - 1 = n - p$ nodes. So the lower p is, the higher the lower-bound, which suggests to look for a partition of the graph into a small number of cliques. Note that a matching can be interpreted as a special case of clique partition: a matching with b edges is equivalent to partitioning the graph into b cliques of size 2, and $n - 2b$ cliques of size 1. In this case, we have $p = n - b$ cliques, hence a lower bound $b = n - p$ as established above.

We are thus driven to look for a minimum clique cover of the graph. Knowing if there exists a clique cover of a given size is NP-complete [Karp, 1972], finding a minimum clique cover is thus NP-hard. Note also that it is NP-hard to approximate the size of a minimum clique cover by a constant factor. Consequently, we resort to heuristics and propose Algorithm 8 as a scalable, greedy one. In short, at each step of the construction of one specific clique, we select a node with the highest priority according to the function ν among candidate nodes. At the first step, any node that is not part of the cover is a candidate, then the candidates must be part of the neighbourhood of all the nodes that have been selected in this particular clique. When no more candidates are available, the clique is maximal: it cannot be extended, so it is added to the partition.

Again, the choice of the priority function ν is critical. We follow a low-degree first strategy, denoted ν_{low} and defined by $\nu_{low}(u) = -x_u$, with x_u the number of neighbours of u that are not in the partition at a given step. The underlying idea is that low-degree nodes are more likely to be grouped together, which reduces the total number of cliques.

Algorithm 8 – Node-greedy heuristic for clique cover

Input: priority function $\nu : V \to \mathbb{Z}$

$\mathcal{P} \leftarrow \emptyset$ ▷ initialise partition
$R \leftarrow V$ ▷ initialise remaining nodes
while $R \neq \emptyset$ **do**
  $K \leftarrow \emptyset$; $C \leftarrow R$ ▷ initialise clique and candidate nodes
  while $C \neq \emptyset$ **do**
    select $u \in C$ with highest priority $\nu(u)$
    $K \leftarrow K \cup \{u\}$; $C \leftarrow C \cap \mathcal{N}_u$ ▷ update clique and candidate nodes
  $\mathcal{P} \leftarrow \mathcal{P} \cup \{K\}$; $R \leftarrow R \setminus K$ ▷ update partition and remaining nodes
return $\mathcal{P}$

The complexity of Algorithm 8 is $\mathcal{O}(n + m)$. Indeed, there are at most n iterations of the main loop. Every time a node u is added to the cover K, each of its edges $\{u, v\}$ with a remaining node v leads to two updates: the priority $\nu(v)$ is updated in constant time, and the belonging of v to candidates C is also checked in $\Theta(1)$.

We can now improve the quality certification using the lower bound given by the smallest clique partition $\mathcal{P}^-$ obtained with Algorithm 8. This leads to the following inequality:

$$n - |\mathcal{P}^-| \leq |C^*| \leq |C_H^-|,$$

where C_H^- is the smallest cover obtained among all solution-heuristics. As any certificate by matching can be rephrased as being obtained from a partition, the following definition is a generalisation of Definition 13:

Definition 14 (Vertex cover certificate) *Given a graph, a vertex cover certificate is a couple $(\mathcal{P}^-, C^-)$, where $\mathcal{P}^-$ is a partition of the nodes into p cliques, and C^- is a vertex cover of s nodes. C^- is certified to be within a factor $\frac{s}{n-p}$ of the minimum cover size and the ratio $\frac{s}{n-p}$ is called the certified quality.*

6.5 Experiments

To evaluate the effectiveness of the quality certification method, we apply it to a collection of real-world networks. The measurements in which we are interested for each network are the certified quality and the execution time needed to obtain them.

6.5.1 Experimental setup

Software and hardware

Regarding state-of-the-art programs, we use the implementation of Hespe et al. [2020] that won the PACE 2019 challenge for exact solutions to the minimum vertex cover problem and call it **ExactVC**. For an approximate solution, we use **FastVC** proposed by Cai et al. [2017]. For the certification method, we run our implementation of the following greedy heuristics, that we release as an open-source c++ repository [2].

We call **GreedyVC** our implementation of node-greedy (Algorithm 7) with highest-degree first priority function, that we use to obtain a small vertex cover. For lower-bounds,

[2] https://github.com/lecfab/certifVC

we call **MatchLB** our implementation of edge-greedy (Algorithm 6) with lowest-degree first priority function, and **CliqueLB** our implementation of the node-greedy heuristic for clique cover (Algorithm 8), both with low-degree first priority functions. All these programs are run on a sgi ice-xa intel xeon e5-2670v3 @2.7 GHz running linux suse 12.3 with 128GB of memory.

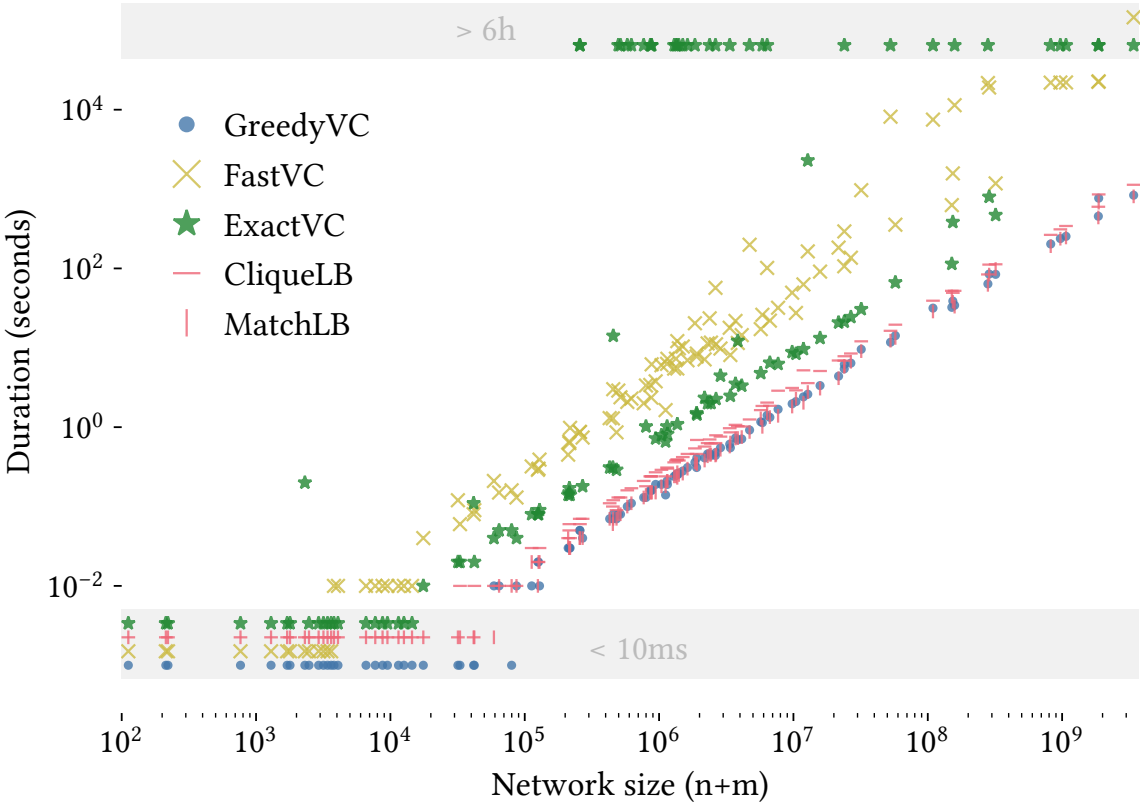


Figure 6.2: **Duration of all considered algorithms with respect to the graph size.** Networks are sorted by increasing size, defined as the sum of their number of nodes and their number of edges $(n + m)$. Executions that take more than six hours or that run out of memory are represented in the top grey zone, while executions that take less than ten milliseconds are considered instantaneous and represented in the bottom grey zone.

Datasets and exhaustive results

To measure the performance of the quality certification method, we apply it on social networks, such as blogs, subgraphs of online social platforms, web graphs such as wikipedia pages, webpages of a linguistic region or top-level domains. More precisely, we use the 114 networks reported in Table 6.1. For comparison purposes, this includes all the undirected graphs of the Network Repository of Rossi and Ahmed [2015] analysed by [Cai et al., 2017], which also includes other types of real-world networks (biological, citation, infrastructure). To test the limits of the different algorithms, we add 10 networks of the *massive* category of Rossi and Ahmed [2015], and two networks of the Webgraph project of Boldi and Vigna [2004]: their names in the table start with a +.

Table 6.1 gives the name of the network and its number of nodes and edges. The next columns display the results of the lower-bound-heuristics CliqueLB and MatchLB, the

Table 6.1: **Results of the experiments on various categories of real-world networks**, sorted by number of edges within each category. The table spans across two pages.

Network	Nodes n	Edges m	Lower-bounds CliqueLB	Lower-bounds MatchLB	Opt ExactVC	Approximate values FastVC	Approximate values GreedyVC	Certified quality
bio-diseasome	516	1,188	285	228	285	285	286	1
bio-yeast	1,458	1,948	456	448	456	456	460	1
bio-celegans	453	2,025	249	224	249	249	256	1
bio-dmela	7,393	25,569	2,628	2,627	2,630	2,632	2,666	1.002
ca-netscience	379	914	214	176	214	214	214	1
ca-CSphd	1,882	1,740	550	550	550	550	556	1
ca-Erdos992	5,094	7,515	461	461	461	461	461	1
ca-GrQc	4,158	13,422	2,207	1,858	2,208	2,208	2,219	1.0005
ca-CondMat	21,363	91,286	12,477	10,121	12,480	12,480	12,510	1.0002
ca-HepPh	11,204	117,619	6,553	5,252	6,555	6,555	6,568	1.0003
ca-AstroPh	17,903	196,972	11,472	8,750	11,483	11,483	11,509	1.001
ca-dblp-2010	226,413	716,460	121,961	99,262	121,969	121,969	122,180	1.0001
ca-citeseer	227,320	814,134	129,188	101,646	129,193	129,193	129,344	1.00004
ca-MathSciNet	332,689	820,644	139,913	128,751	139,951	139,951	140,605	1.0003
ca-dblp-2012	317,080	1,049,866	164,928	137,185	164,949	164,949	165,229	1.0001
ca-coauthors-dblp	540,486	15,245,729	472,090	269,867	472,179	472,179	472,362	1.0002
ca-hollywood-2009	1,069,126	56,306,653	863,973	533,909	864,052	864,052	864,219	1.0001
ia-enron-only	143	623	83	70	86	86	87	1.036
ia-infect-hyper	113	2,196	85	56	90	90	93	1.059
ia-infect-dublin	410	2,765	288	205	293	294	296	1.021
ia-email-univ	1,133	5,451	584	547	594	594	603	1.017
ia-fb-messages	1,266	6,451	574	572	578	578	594	1.007
ia-reality	6,809	7,680	81	81	81	81	81	1
ia-email-EU	32,430	54,397	820	819	820	820	820	1
ia-enron-large	33,696	180,811	12,771	10,777	12,781	12,781	12,806	1.001
ia-wiki-Talk	92,117	360,767	17,288	17,263	17,288	17,288	17,417	1
inf-power	4,942	6,548	2,185	2,179	2,186	2,187	2,255	1.001
inf-roadNet-PA	1,087,562	1,541,514	535,759	523,223	†	555,290	584,931	1.036
inf-roadNet-CA	1,957,027	2,760,388	965,008	941,868	†	1,001,279	1,054,981	1.038
inf-road-usa	23,947,347	28,854,312	11,261,882	11,142,934	†	11,529,731	12,092,364	1.024
rec-amazon	91,813	125,704	47,414	42,838	47,605	47,607	49,005	1.004
rt-retweet	96	117	32	32	32	32	33	1
rt-twitter-copen	761	1,029	237	233	237	237	239	1
rt-retweet-crawl	1,112,702	2,278,852	81,040	81,037	81,040	81,048	81,345	1.0001
sc-nasasrb	54,870	1,311,227	50,772	27,434	†	51,258	51,657	1.010
sc-shipsec1	140,385	1,707,759	112,486	70,189	†	117,341	119,590	1.043
sc-shipsec5	179,104	2,200,076	142,864	89,520	†	147,170	148,882	1.030
sc-pkustk11	87,804	2,565,054	83,886	43,902	83,911	83,913	84,155	1.0003
sc-pkustk13	94,893	3,260,967	88,540	47,445	†	89,230	89,690	1.008
sc-pwtk	217,891	5,653,221	207,279	108,945	†	207,725	208,478	1.002
sc-msdoor	404,785	9,378,650	381,408	202,379	381,558	381,559	382,142	1.0004
sc-ldoor	909,537	20,770,807	856,631	454,742	856,754	856,758	858,166	1.0001
web-polblogs	643	2,280	243	240	244	244	246	1.004
web-google	1,299	2,773	498	405	498	498	498	1
web-edu	3,031	6,474	1,451	1,410	1,451	1,451	1,563	1
web-BerkStan	12,305	19,500	5,248	4,709	5,384	5,389	5,483	1.027
web-webbase-2001	16,062	25,593	2,645	2,325	2,651	2,652	2,684	1.003
web-spam	4,767	37,375	2,275	2,126	2,297	2,298	2,331	1.010
web-indochina-2004	11,358	47,606	7,300	5,121	7,300	7,300	7,395	1
web-sk-2005	121,422	334,419	57,503	44,022	58,173	58,176	58,510	1.012
web-arabic-2005	163,598	1,747,269	114,383	70,119	114,420	114,430	115,316	1.0004
web-wikipedia2009	1,864,433	4,507,315	645,457	626,961	†	648,343	658,411	1.004
web-it-2004	509,338	7,178,413	414,492	227,286	414,507	414,676	415,137	1.0004
web-uk-2005	129,632	11,744,049	127,774	64,590	127,774	127,774	127,774	1

Network	Nodes n	Edges m	Lower-bounds CliqueLB	Lower-bounds MatchLB	Opt ExactVC	Approximate values FastVC	Approximate values GreedyVC	Certified quality
soc-karate	34	78	14	13	14	14	14	1
soc-dolphins	62	159	34	30	34	34	35	1
soc-wiki-Vote	889	2,914	404	401	406	406	413	1.005
soc-epinions	26,588	100,120	9,752	9,545	9,757	9,757	9,847	1.001
soc-brightkite	56,739	212,945	21,174	20,765	21,190	21,190	21,448	1.001
soc-douban	154,908	327,162	8,685	8,685	8,685	8,685	8,695	1
soc-slashdot	70,068	358,647	22,368	22,160	22,373	22,373	22,604	1.0002
soc-twitter-follows	404,719	713,319	2,323	2,323	2,323	2,323	2,323	1
soc-gowalla	196,591	950,327	83,988	81,089	84,222	84,224	85,252	1.003
soc-delicious	536,108	1,365,961	85,290	85,278	85,298	85,999	87,634	1.008
soc-youtube	495,957	1,936,748	146,306	144,725	146,376	146,376	148,004	1.0005
soc-BlogCatalog	88,784	2,093,195	20,748	20,647	20,752	20,752	20,951	1.0002
soc-LiveMocha	104,103	2,193,083	43,396	43,294	43,427	43,429	44,060	1.001
soc-buzznet	101,163	2,763,066	30,477	30,138	30,613	30,626	30,993	1.005
soc-youtube-snap	1,134,890	2,987,624	276,916	274,303	276,945	276,945	278,998	1.0001
soc-flickr	513,969	3,190,452	153,048	148,997	153,271	153,272	154,384	1.001
soc-FourSquare	639,014	3,214,986	90,099	89,800	90,108	90,110	90,570	1.0001
soc-lastfm	1,191,805	4,519,330	78,688	78,668	78,688	78,688	78,962	1
soc-digg	770,799	5,907,132	103,230	102,898	103,234	103,246	104,337	1.0002
soc-flixster	2,523,386	7,918,801	96,317	96,298	96,317	96,317	96,439	1
soc-pokec	1,632,803	22,301,964	821,754	780,762	†	843,444	856,756	1.026
soc-livejournal	4,033,137	27,933,062	1,858,242	1,775,169	1,868,903	1,869,052	1,890,878	1.006
soc-orkut	2,997,166	106,349,209	1,953,415	1,485,585	†	2,170,950	2,208,766	1.111
socfb-CMU	6,621	249,959	4,735	3,301	†	4,987	5,049	1.053
socfb-MIT	6,402	251,230	4,436	3,187	†	4,658	4,723	1.050
socfb-UCSB37	14,917	482,215	10,576	7,449	†	11,266	11,442	1.065
socfb-Duke14	9,885	506,437	7,195	4,931	†	7,685	7,794	1.068
socfb-Stanford3	11,586	568,309	8,087	5,764	†	8,518	8,602	1.053
socfb-UConn	17,206	604,867	12,305	8,592	†	13,235	13,436	1.076
socfb-UCLA	20,453	747,604	14,299	10,208	†	15,230	15,460	1.065
socfb-OR	63,392	816,886	35,593	30,930	†	36,553	37,131	1.027
socfb-Wisconsin87	23,831	835,946	17,175	11,904	†	18,396	18,665	1.071
socfb-Berkeley13	22,900	852,419	16,146	11,426	†	17,221	17,488	1.067
socfb-UIllinois	30,795	1,264,421	22,505	15,379	†	24,103	24,475	1.071
socfb-Indiana	29,732	1,305,757	21,599	14,852	†	23,323	23,664	1.080
socfb-Penn94	41,536	1,362,220	29,137	20,743	†	31,176	31,669	1.070
socfb-UF	35,111	1,465,654	25,483	17,546	†	27,319	27,745	1.072
socfb-Texas84	36,364	1,590,651	26,011	18,169	†	28,186	28,585	1.084
socfb-B-anon	2,937,612	20,959,854	303,048	302,989	303,048	303,049	303,574	1.000003
socfb-A-anon	3,097,165	23,667,394	375,230	375,086	375,230	375,233	376,158	1.00001
socfb-uci-uni	58,790,782	92,208,195	866,766	866,765	866,766	866,768	867,207	1.000002
tech-routers-rf	2,113	6,632	795	782	795	795	806	1
tech-as-caida2007	26,475	53,381	3,683	3,679	3,683	3,683	3,694	1
tech-WHOIS	7,476	56,943	2,281	2,193	2,284	2,284	2,298	1.001
tech-internet-as	40,164	85,123	5,699	5,685	5,700	5,700	5,716	1.0002
tech-p2p-gnutella	62,561	147,878	15,682	15,682	15,682	15,682	15,727	1
tech-RL-caida	190,914	607,610	74,320	73,030	74,593	74,936	75,596	1.008
tech-as-skitter	1,694,616	11,094,209	523,872	511,379	525,022	527,186	529,663	1.006
+tech-p2p	5,792,297	147,829,887	301,717	301,716	301,717	301,718	304,450	1.000003
+web-indochina-2004-all	7,414,758	150,984,819	2,687,877	2,202,789	†	2,720,219	2,757,123	1.012
+soc-sinaweibo	58,655,849	261,321,033	223,000	223,000	223,000	223,000	223,171	1
+web-uk-2002-all	18,483,186	261,787,258	6,533,448	5,621,934	†	6,627,415	6,684,896	1.014
+soc-twitter-2010	21,297,772	265,025,545	7,613,876	7,415,374	7,645,886	7,646,009	7,737,666	1.004
+web-uk-2005-all	39,454,463	783,027,125	15,624,892	12,692,243	†	15,854,309	15,952,280	1.015
+web-webbase-2001-all	115,554,441	854,809,761	37,941,623	33,519,632	†	38,558,811	38,896,967	1.016
+web-it-2004-all	41,290,648	1,027,474,947	15,555,070	13,091,060	†	15,815,492	15,988,487	1.017
+soc-friendster	65,608,366	1,806,067,135	28,938,176	28,075,409	†	29,304,576	29,614,049	1.013
+web-sk-2005-all	50,636,059	1,810,063,330	19,535,660	16,381,409	†	20,447,574	20,353,317	1.042
+webgraph-twitter-2010	41,652,230	1,202,513,046	12,828,884	12,449,801	†	12,906,788	13,065,672	1.006
+webgraph-uk-2007-05	105,153,952	3,301,876,564	38,243,390	32,147,185	†	†	39,395,634	1.030

exact size of a minimum vertex cover found with ExactVC, and the results of the solution-heuristics FastVC and GreedyVC. The symbol † marks executions that did not finish in the allocated 6 hours. The last column shows the certified quality (see Definition 14), which is the ratio between the lowest result of solution-heuristics and the highest result of lower-bound-heuristics; it is between 1 and 2, where 1 is a desired proof of optimality while 2 is the worst-case theoretical guarantee.

6.5.2 Scalability

Figure 6.2 shows the runtime of the heuristics listed above as well as the exact solution. Note that we impose a 6 hours time limit for all experiments.

We report several observations. First, the running of GreedyVC, MatchLB and CliqueLB is almost linear as expected, and these heuristics appear to be the fastest among all studied methods.

Second, we observe that ExactVC cannot always find a solution within the allowed time limit (6 hours). It does on 79 network instances, that we then call *solved networks* (several of which have millions of nodes) but fails on 35 *unsolved networks*, where the optimum remains unknown. For solved networks, ExactVC runs almost as fast as GreedyVC and seems to scale linearly. Third, the FastVC heuristic, which is the best solution heuristic available, is able to find a solution within the allocated time for all networks except one (where it runs out of memory). Notice that it is significantly slower than the other algorithms, though it does manage to find an approximate solution in most cases where ExactVC does not.

For the sake of completeness, we also ran experiments for the blossom algorithm using the implementation of the boost[3] c++ library.

These experiments showed that the lower bounds obtained with blossom are only slightly better than the ones obtained with MatchLB, and far worse than the ones obtained with CliqueLB, which makes this heuristic irrelevant for the purpose of this work, thus we do not report these results.

6.5.3 Practical certification of the minimum vertex cover

We present the results of the certification obtained by using MatchLB as a lower-bound-heuristic, and the solution-heuristics which performs best in Figure 6.3. We observe that the certifications obtained are good for a significant number of networks, in the sense that for approximately half of them, our approach is able to guarantee that the obtained value is within a factor 1.1 of the optimal value. However, the certification remains quite poor in some cases, and can even be as bad as the factor 2 which corresponds to the worst-case theoretical guarantee. In those cases, using a large matching as a lower bound does not bring additional insight on the quality of the results. As explained in Section 6.4.3, this is due to the fact that this approach is inefficient when a network contains many large cliques.

To verify this, we also indicate in Figure 6.3 the improvement of the bound obtained when the clique cover heuristics is used for the lower bound instead. We can see that this lower bound provides much better results than large matching. Also, when comparing the results on the solved instances to the ones that are not, we can see that in general the certifications obtained for networks which are not solved exactly tend to be of slightly lower quality.

[3] https://boost.org/doc/libs/1_80_0/libs/graph/doc/maximum_matching.html

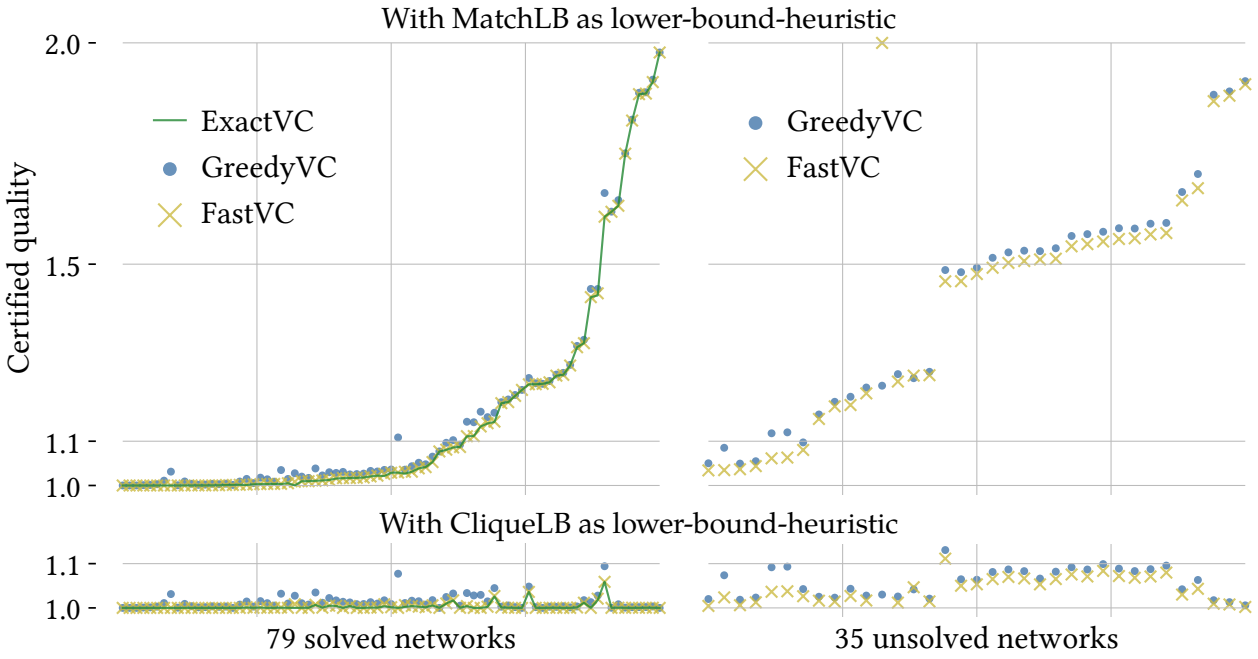


Figure 6.3: **Certifed quality for vertex cover.** We show the certified quality for solution-heuristics GreedyVC and FastVC, as well as ExactVC when available, for solved (left charts) and unsolved networks (right charts). The lower-bound-heuristic used for the quality certification is either MatchLB (top charts) or CliqueLB (bottom charts). In each column, networks are sorted according to their certified quality with MatchLB, which allows to see the improvement provided by the CliqueLB lower-bound.

To have a better view of the quality certifications obtained using the best solution-heuristic (which is either FastVC or GreedyVC) and the best lower-bound-heuristic (which is always CliqueLB), we show them on another vertical scale in Figure 6.4 (left). The certified quality is under 1.11 for all networks, and under 1.01 for 79 of them. Note that even among unsolved networks, 15 of them have a certified quality under 1.03: while the exact minimum is unknown, a solution-heuristic found a cover that is guaranteed to be at most 3% larger than the minimum, thanks to a lower-bound found by CliqueLB. Altogether these results indicate that the quality certification method is an efficient way to prove that the results of solution-heuristics are very close to the optimum.

Still, we are interested in knowing where there is room for improvement. In Figure 6.4 (right), we represent for the solved networks the ratio between the upper-bound obtained with the best solution-heuristic and the exact result, and the ratio between the lower-bound obtained with the best lower-bound-heuristic and the exact result. We can see that the ratio of the upper-bound to the exact result is strikingly close to 1 in nearly all cases. This means that in a majority of cases, the quality of the certification method is limited by the lower-bound-heuristic, which could be expected, as much more effort has been made to create efficient solution-heuristic in the literature. So, we think that there is the potential to enhance the certification by improving the lower-bound-heuristic.

In any case, CliqueLB is a fast and scalable method that can be used for quality certification with low extra computational cost.

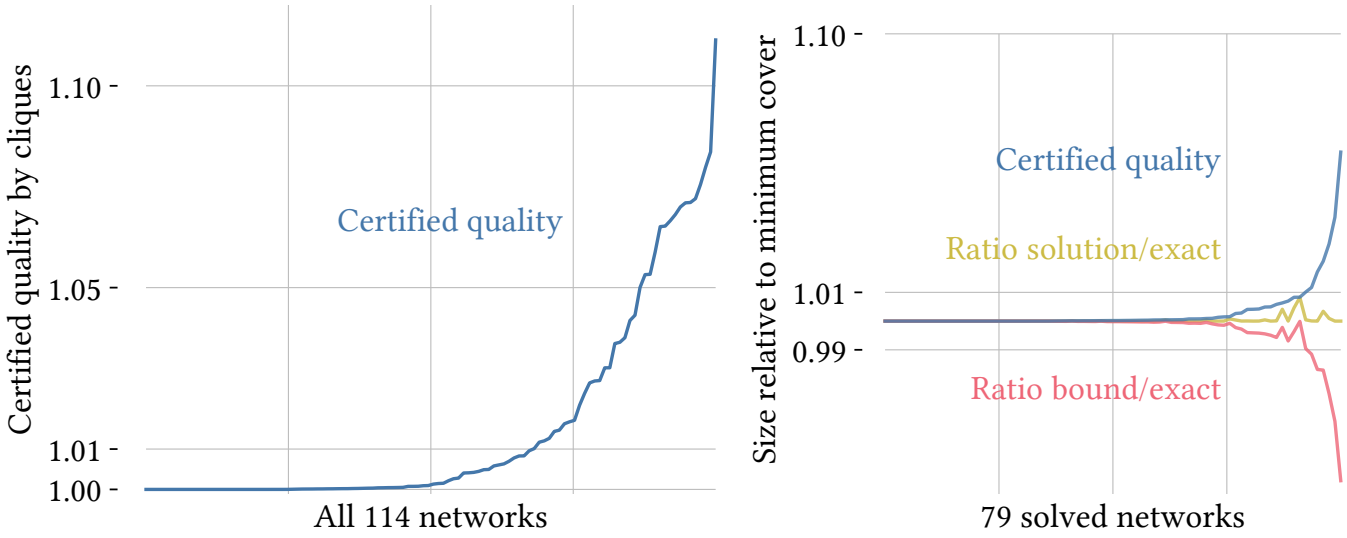


Figure 6.4: Left: Certified quality for vertex cover the using best performing solution- and lower-bound-heuristics. Right: Certified quality for solved networks, shown with the gap due to solution-heuristics (ratio between the smallest solution-heuristic result and the exact result) and the gap due to lower-bound-heuristics (ratio between the largest lower-bound-heuristic result and the exact result).

6.6 Certifying related problems

While the focus of this work is on the quality certification of vertex cover, other problems can be certified with the same results. Let us first consider the minimum clique cover. As mentioned in Section 6.3, it is NP-hard to obtain an exact solution or even to approximate it within a constant-factor. However, we have seen that the following inequality stands:

$$n - p \leq n - p^* \leq c^* \leq h$$

where n is the number of nodes, p is the size of a partition of the nodes into cliques and p^* the minimum size (or the result of the minimum clique cover problem), h is the size of a vertex cover found by a solution-heuristic and c^* is the minimum size of a vertex cover. As p and h can be obtained experimentally, we obtain a certification by rewriting the previous inequality into $n - h \leq p^* \leq p$. In other word, it gives a certified quality ratio for the clique cover p:

$$p \leq \frac{p}{n-h} p^*$$

Another problem that benefits from this certification is the maximum independent set problem. As the complement set of a vertex cover is an independent set, the solution s^* for the independent set problem satisfies $s^* = n - c^*$. The above inequality can be rewritten $n - p \leq n - p^* \leq n - s^* \leq n - s$ where s is the size of the largest independent set found by heuristics. Thus, $p \geq s^* \geq s$, which gives a certified quality ratio for the independent set s:

$$s \leq \frac{s}{p} s^*$$

Note that this ratio is between 0 and 1 as independent set is a maximisation problem.

More generally, the quality certification can apply on all problems that have a constant-factor approximation – like vertex cover – or that can be expressed as a transformation of

problems that have one – like independent set, but also on problems that have theoretical and empirical bounds. Experimental studies are necessary to assess whether these bounds give valuable certificates on real-world instances.

6.7 Conclusion

This work proposes a practical method to certify the quality of approximate solutions to optimisation problems. It consists in using jointly a heuristic to approach the optimum, and another to provide a bound of the optimum. We illustrate on the NP-hard problem of finding a minimum vertex cover, for which we propose an adequate choice of heuristics: one is a state-of-the-art implementation that finds a small vertex cover, and the other is a greedy algorithm for finding a small clique cover. The resulting certified quality is much better than the best existing theoretical factor 2, even on networks with billions of edges where obtaining an exact solution is costly or unfeasible.

As we suggested in Section 6.5.3, designing better lower-bound-heuristics would allow for further improvements of the vertex cover certification. Beyond vertex cover, we believe that many other problems could benefit from the certification method and hope that this work will incentivise efforts in this direction. A first direction could consist in applying a quality certification on the heuristics of Chapter 5: Algorithm 4 provides a lower-bound for the cost C^{++}, which can serves to certify the quality of a heuristic such as core ordering. Designing tighter bounds or more efficient heuristics, and addressing the exact optimisation of the cost, may give better insights on these problems.

Chapter 7

Network science

Mobility patterns in the scientific landscape

Summary

This chapter is an addition to the thesis and does not involve node orderings. It is a joint work with the Interaction Data Lab [1], whose researchers use network approaches on large datasets to study collaboration in various contexts: the evolution of science through publications, the development of open-source programs with thousands of contributors, or the projects of citizen science. The aim of this collaboration was to apply the algorithmic tools studied in the rest of the thesis to concrete questions of data analysis. Network science seemed like an ideal field to apply concepts of experimental graph algorithmics. Integrating this team was a crucial experience of inter-disciplinary collaboration, and it led to interesting discussions while confronting the views of people with different backgrounds. In the end, we chose to set aside the graph description of our dataset in order to work with embeddings, a more usual tool in the science of science domain.

We address the following question: considering knowledge as a vast space to explore, how do researchers move throughout their career? To answer it, we use formal methods to systematically examine patterns of scientific knowledge mobility. First, we create a model of the knowledge space: based on more than one million papers shared on the arXiv platform, we define a high-dimensional space that indicates the scientific field of each paper, and we project it into two dimensions using embedding techniques. Second, we analyse the publication history of each individual researchers as a trajectory in the knowledge space and find that their mobility patterns closely resemble physical mobility patterns: mobility flows are well described by a gravity model, where researchers are more likely to move towards areas of high density and less likely to travel longer distances. Finally, we identify two types of researchers based on their individual mobility patterns: *explorers*, who tend to explore wider portions of the space and develop new research areas, and *exploiters*, who focus on the exploitation of one zone to develop expertise. Our findings suggest that spatial mobility analysis is a valuable tool for understanding how knowledge evolves over time, and how researchers navigate and contribute to this evolution.

Contributions

* Build a representation of the knowledge space using a low-dimensional embedding of arXiv dataset.
* Exploit metrics derived from human mobility to describe research trajectories.
* Show that scientific mobility in the knowledge space follows a gravity model, a parallel with what is observed in spatial mobility.
* Categorise researchers between explorers and exploiters.

Publication

* *Charting mobility patterns in the scientific knowledge landscape*
  Singh, Tupikina, Lécuyer, Starnini, and Santolini [2023], submitted EPJ DataScience.

[1] https://interactiondatalab.com

7.1 Introduction

Quantifying the evolution of knowledge is crucial to understanding the past and predicting future innovations [Belikov et al., 2022], which ultimately leads to societal progress. At the forefront of scientific innovation are researchers recombining ideas to push the boundaries of the known [Iacopini et al., 2018, Ferreira et al., 2020]. With the exponential growth in the number of authors and publications [Bornmann et al., 2021, Fortunato et al., 2018], novel methods are needed to represent and provide insights into knowledge development.

The increasing access to large-scale publication datasets has provided opportunities to quantify the choices made by researchers and examine the factors governing the evolution of knowledge. By studying the citation patterns of researchers in their publications, studies have measured how conflicting ideas are pursued by researchers before they converge to a common consensus [Shwed and Bearman, 2010] or give way to new ideas [Lin et al., 2022]. Other studies have focused on identifying "hot topics" in research [Liu et al., 2018], quantifying knowledge flow patterns [Sun and Latora, 2020] and memory effects in the evolution of knowledge [Yin and Wang, 2017, Pan et al., 2018], or predicting the ultimate impact of a researcher [Wang et al., 2013, Sinatra et al., 2016]. Similarly, keywords and phrases from publications can be leveraged to track the evolution of scientific ideas and fields [Chavalarias and Cointet, 2013, Battiston et al., 2019] or quantify how scientists choose and shift their research focus over time [Jia et al., 2017, Zeng et al., 2019]. For example, the Physics and Astronomy Classification Scheme (PACS) used in articles published by the American Physical Society can be exploited to study the "essential tension"

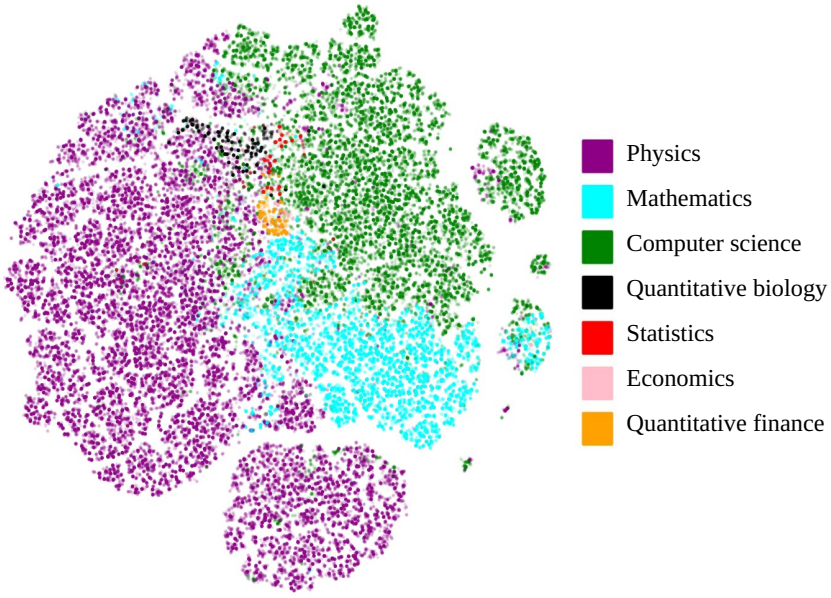


Figure 7.1: **Construction of the knowledge space.** We use the metadata from 1.45 million articles posted on the arXiv, corresponding to the article field tags, authors, and timestamp. We build a high-dimensional 175 space where each article is uniquely mapped through field tags corresponding to orthogonal dimensions. This high-dimensional space is finally embedded within a 2-dimensional knowledge space using the tSNE algorithm. Each point represents an article. Colours correspond to major academic fields in arXiv based on the first tag (i.e primary field) of the articles.

between exploring the boundaries of a research area and exploiting previous work [Aleta et al., 2019]. Finally, scientific credit among researchers and their mutual scientific interest (quantified by citations between papers and keywords, respectively) can be combined to improve the prediction of new scientific collaborations [Tuninetti, Marta et al., 2021].

Therefore, studying the publication trajectories of researchers can help identify the multifaceted and complex processes underlying the evolution of knowledge. Such trajectories are often talked about metaphorically, for example when referring to some scientific advances as "great leaps" [Holden, 1974]. Here, we aim to explore the parallel between scientific and human mobility more formally, by leveraging insights from human mobility studies. Using large-scale real-world data on human trajectories, previous studies have indeed uncovered several laws underlying human mobility. Despite heterogeneity in their movement, humans exhibit recurring patterns in their mobility [Wu et al., 2021, Ubaldi et al., 2021]. These patterns have been shown to give rise to scaling laws for the travel distance distribution [Barbosa et al., 2018]. At the macroscopic level, the resulting flows between two locations follow a gravity model [Schläpfer et al., 2021], mimicking the Newtonian law of attraction between two masses at a given distance. Beyond jump distance, individuals show reproducible properties at the whole trajectory level. For example, individuals can be categorised into two classes, *returners* and *explorers*, depending on their propensity to come back to the same location or explore new ones [Pappalardo et al., 2015]. More generally, studies on both individual and collective mobility datasets have proposed various quantitative models explaining the dynamics of human mobility [Simini et al., 2021, Alessandretti et al., 2020, Barbosa et al., 2018, Schneider et al., 2013, Simini et al., 2012, Wilson, 1967]. Crucially, these reproducible patterns are not unique to human mobility [Hills et al., 2015]. Multiple studies across disciplines have found striking similarities between human mobility in geographic space and animals foraging [Hills, 2006], insects swarms [Bonabeau et al., 1999], search methods in abstract environments such as memory space [Hills et al., 2012], organisational learning [March, 1991], and cyberspace [Zhao et al., 2014, Hu et al., 2018, Barbosa et al., 2016].

In the context of knowledge evolution, tools and data sources now abound for spatial representations. Natural language processing and embedding methods with metadata from publications such as citations, keywords, or abstracts, can be combined to exploit similarities between research publications and derive a low-dimensional representation of the scientific landscape. Such representations have been used to quantify the extent of ideas explored by researchers [Milojević, 2015, Milojević et al., 2011] and the structuration of journals [Peng et al., 2021], giving insights into the structure of knowledge.

In this work, we solidify these intuitions into a quantitative framework to represent scientific knowledge and exploit metrics derived from human mobility to describe research trajectories. We use embedding methods on publication metadata from arXiv pre-prints to build a low-dimensional *knowledge space* [Ying et al., 2015]. We then track the mobility of disambiguated authors in this space using their publication records. We find that knowledge exploration shows striking similarities with human mobility in physical space. First, we show that scientific mobility in the knowledge space follows a gravity model, with jumps more likely to occur in areas of high density and less likely to occur over longer distances. Second, we retrieve a dichotomy in knowledge exploration between interdisciplinary scientific *explorers* – more likely to disrupt and pioneer new fields – and *exploiters*, who tend to exploit a particular area of expertise, mirroring what is observed in spatial mobility between explorers and returners. Finally, we discuss the usefulness of knowledge mobility analyses for the study of science and innovation, and discuss limitations and implications for future works.

7.2 Results

7.2.1 Scientific trajectories in the arXiv knowledge space

To build the knowledge space, we leverage the arXiv dataset, encompassing $1,456,403$ scientific articles published online between 1992 and 2018 (see Methods and the previous work of Singh et al. [2022]). Our interest in this dataset is two-fold. First, it has a clear and stable ontology for field tags, which are used by authors to specify the relevant research area(s) covered by their articles. There is a strong incentive for authors to document these tags as precisely as possible, in order for their article to appear in the right arXiv section searched by the target scientific community, and in the relevant daily email digest that interested scientists can subscribe to. Second, as a pre-print server, it has no editorial barrier or publication cost, creating a low threshold for publication. This allows us to track the publication history of an author in a fine-grained manner, at the time they are considered finished, and irrespective of their perceived novelty. As such, arXiv pre-prints can be thought of as tracking *knowledge steps* to a high resolution, without requirements for novelty thresholds to be met.

We first build a spatial representation of the knowledge space formed by arXiv pre-prints (Figure 7.1). The structure of this space is determined by the 175 tags used by submitting authors to assign scientific sub-fields to articles. Articles can be assigned with one or more tags. For instance, an article can be tagged with Social and Information Networks (cs.si) and Physics and Society (physics.soc-ph). An article can thus be represented as a binary vector $X = (0, 0, 1, 0, 1, ..., 0)$ in the high dimensional 175 sub-fields space, with $X_i = 1$ if the article has the tag corresponding to scientific field i.

Since articles rarely combine more than a few tags (see Figure 7.2), the knowledge space is sparsely populated. Moreover, some tags co-occur frequently, creating redundant information [Singh et al., 2022]. Following these observations, we reduce the dimensionality of this initial space by embedding it into a low-dimensional space via the tSNE algorithm [Van der Maaten and Hinton, 2008, Van der Maaten, 2009] (see Methods). In this study, we focus on a two-dimensional embedding to match traditional studies of human geographical mobility. In addition, we discuss the stability of the results with other embedding approaches in the Methods section. Figure 7.1 shows the resulting *knowledge*

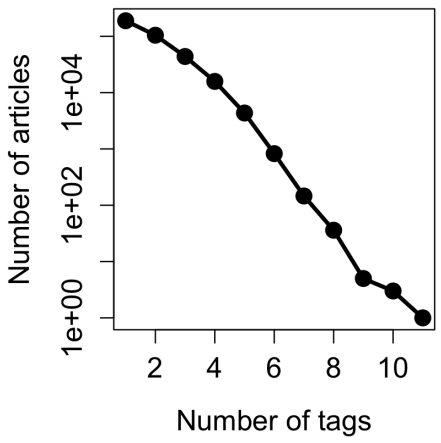


Figure 7.2: **Sparsity of the high dimensional space.** Number of articles in arXiv with a given number of tags. Most papers have less than 10 tags, which is much less than the dimension of the space ($N = 175$): most possible locations in the space are not populated.

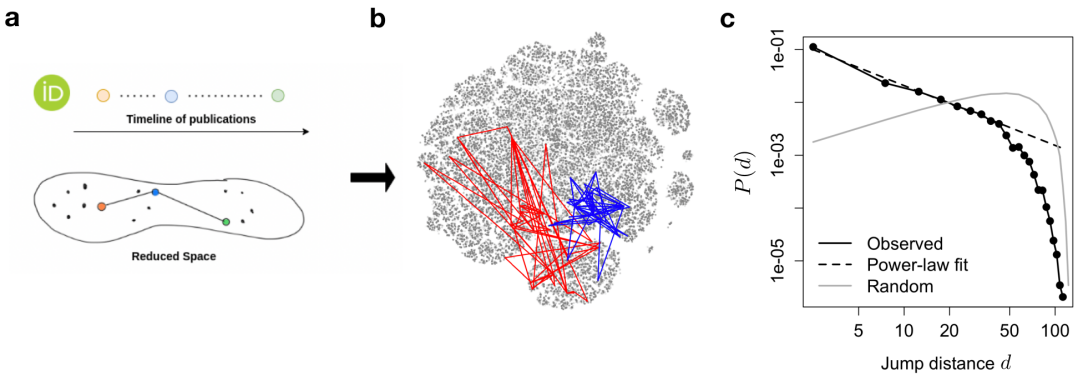


Figure 7.3: **Scientific mobility in the knowledge space. a.** The sequence of arXiv preprints of each researcher identifies a unique scientific trajectory in the knowledge space. **b.** We show two example trajectories indicating different behaviours of researchers. **c.** Distribution of consecutive jump distances for authors with at least 10 publications ($N = 11,826$). The dashed black line is a guide for the eye indicating a power-law behavior. The gray line corresponds to the distribution of jump distances obtained when locations are selected at random across all possible visited locations, for each author.

space, where articles are represented as points coloured according to their primary (first) field tag. We observe that articles belonging to the major fields from arXiv cluster into distinctive well-defined regions of the space, with interdisciplinary fields such as Quantitative Biology (q-bio) or Quantitative Finance (q-fin) located at the interface between related disciplines.

The chronological sequence of articles published by an author defines a sequence of locations in the knowledge space, tracing their *scientific trajectory* (Figure 7.3a-b). In order to obtain high-quality trajectories, we select a sample of $11,826$ from a total of $50,402$ disambiguated researchers for which we have a unique ORCID identifier, and who published at least 10 articles. Within a trajectory, two consecutive articles constitute a jump, with a length equal to the (euclidean) distance computed in the embedding, and duration equal to the number of days elapsed between the two articles. If the authors were randomly jumping across all possible locations in the space, the jump distribution would follow a bounded distribution around a typical, large step size (Figure 7.3c, gray line), according to a pure diffusive process. Instead, Figure 7.3c shows that the jump distance distribution is compatible with a power-law functional form, with a cut-off at large distances due to the finite size of the space, differing significantly from a diffusive process. Importantly, we observe that this feature is robust with respect to different embedding techniques, see Figure 7.12. This indicates that, while the majority of jumps are small, with researchers orbiting relatively close to a particular research interest, a small fraction of jumps extend far into the knowledge space, standing for researchers crossing fields. In the next section, we investigate whether simple models of human mobility can be compatible with the observed behaviour.

7.2.2 A gravity model of scientific mobility

The observed fat-tail form (with a cut-off) of the jump size distribution is reminiscent of the inverse relation with distance observed in human mobility flows between two locations. This observation led to a simple and intuitive model in human mobility studies, the gravitation model, where the flux F_{ij} between two locations i and j is proportional to the

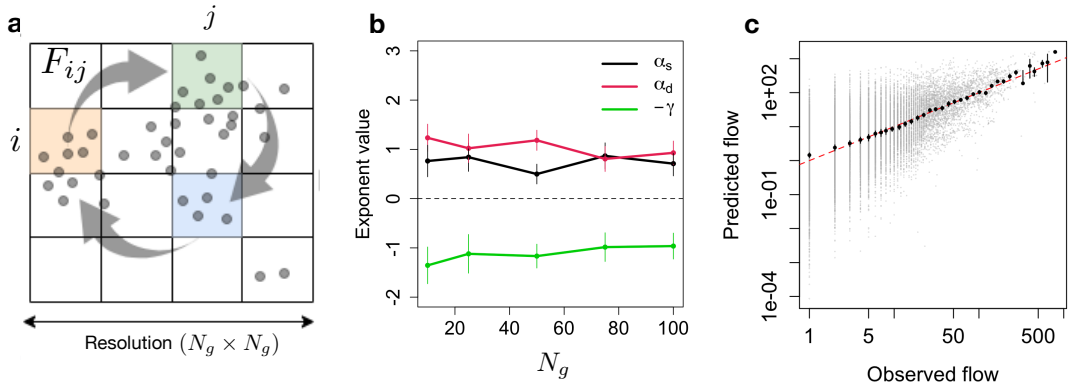


Figure 7.4: **A gravity model for scientific mobility. a.** By dividing the projected space into a grid, we can calculate aggregated densities and study their effect on flow patterns between grid elements. The resolution of the grid is fixed by the number of cells along any dimension N_g. **b.** Fitted exponents from Eq. (7.1) for different grid resolution levels $N_g = 10, 25, 50, 75, 100$. **c.** Comparison between predicted and observed mobility flows at a grid size resolution $N_g = 10$.

population sizes at i and j and inversely proportional to the distance d_{ij} between them. Earlier works on spatial distribution models and urban modelling [Wilson, 1967, Senior, 1979, Wilson, 2013] have shown that such a model can be functionally derived from statistical mechanics insights and empirical laws such as Zipf's law [Ribeiro and Rybski, 2021]. When considering population-scale mobility in an origin-destination setting such as ours, the gravity model naturally emerges as the expectation of the distribution maximising the entropy of mobility between two locations [Wilson, 1967].

Much like the urban vs rural landscape, where populations conglomerate into a few, dense regions corresponding to urban areas, there are denser regions in the low-dimensional knowledge space, corresponding to more investigated areas. However, unlike cities and administrative areas, we do not have a clear definition of boundaries in the knowledge space. Here, we use a simple box/container model by defining a grid of size $N_g \times N_g$ covering the knowledge space, where N_g is a parameter quantifying the resolution level, and population counts are aggregated at the grid level (Figure 7.4a). We then define a gravity model to predict the observed flow F_{ij} between two grid locations i and j in the knowledge space, defined as the number of scientists jumping from grid location i to location j, by using a rolling time window of 5 years:

$$\tilde{F}_{ij} = G \frac{V_i^{\alpha_s} V_j^{\alpha_d}}{d_{ij}^{\gamma}}, \tag{7.1}$$

where $\tilde{F}_{ij}$ is the predicted flow between locations i and j, G is a normalisation constant, d_{ij} is the distance between locations i and j, and V_i and V_j (visits) are the numbers of authors who have published an article in locations i and j during the 5 previous years. The exponents α_s, α_d, and γ introduce non-linear scalings, such as crowding effects for the number of visits, where higher densities lead to sublinear ($\alpha < 1$) or superlinear ($\alpha > 1$) increase in flow.

Figure 7.4b shows the values of the exponents obtained by fitting Eq. (7.1) to the empirical flows F_{ij} at different resolution levels N_g (see Methods). Overall, we find a remarkable stability across grid sizes, with coefficients close to 1. The quality of fit is shown in Figure 7.4c, comparing predicted flows with observed flows, with a Pearson correlation

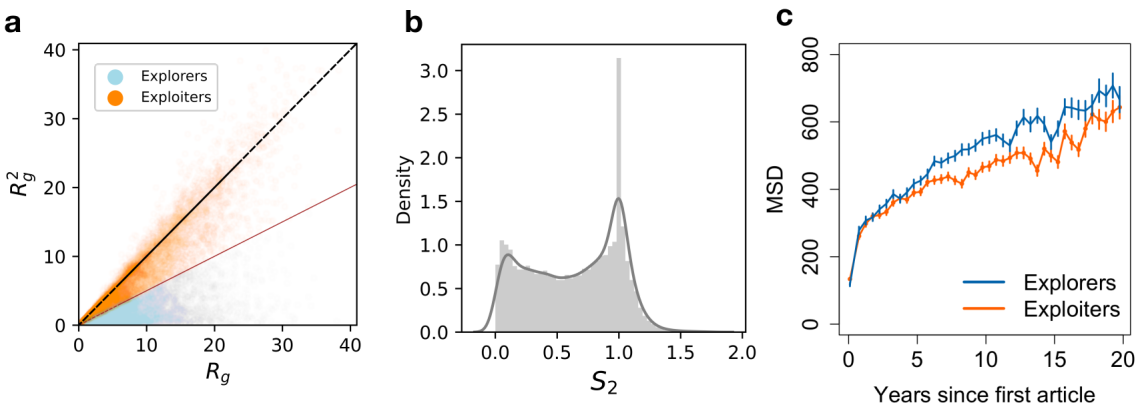


Figure 7.5: **Explorers and exploiters in the knowledge space. a.** Comparison of the radius of gyration R_g^2, with the centre of mass computed from the 2 most visited locations, with the full radius of gyration R_g (see Methods). We find a dichotomy between exploiters (orange) and explorers (blue), using the bisector method to classify them [Pappalardo et al., 2015]. **b.** Distribution of $S_2 = R_g^2/R_g$, further highlighting the dichotomy as a bimodality of exploiters (close to 1) and explorers (close to 0). **c.** Comparison of the mean squared displacement as a function of time since their first article for explorers and exploiters.

value of $r = 0.58$, indicating that the model explains $r^2 = 33.6\%$ of the variance of the mobility flows in the knowledge space. We note that the observed correlation is larger than the ones observed for real-world mobility (with r between $0.03 - 0.49$, see [Simini et al., 2021]). Finally, we find that beyond tSNE, the gravity model is able to represent flows of scientific mobility for different embedding techniques, with qualitatively similar exponents (see Figure 7.13).

7.2.3 Scientific explorers vs exploiters

When jumping to their next article, researchers can move to a novel region of the space, or return to a previous one. That is, in our framework, researchers choose between exploring a new scientific field or exploiting the previous knowledge that they built. While such behaviours can lead to similar jump distribution patterns, they will impact more general statistics about the full trajectory, such as the extent of spatial territory covered. Previous studies have uncovered such a heterogeneity between individual trajectories in human mobility patterns, highlighting a dichotomy between returners, who gravitate around a small number of locations, and explorers, who rather move to new locations. These results have been found to hold both for spatial [Pappalardo et al., 2015], as well as virtual [Barbosa et al., 2018] contexts. Here, we explore whether such a heterogeneity exists in the context of knowledge exploration.

To assess the extent of territory covered by a trajectory, we study the radius of gyration R_g, defined as the average distance of visited locations to their centre of mass (see Eq. (7.3) in Methods). By limiting to the top k most visited locations, one can define the corresponding radius of gyration R_g^k (Eq. (7.4)) and compare it with the full R_g to evaluate the extent to which the trajectory returns to k locations. Figure 7.5a shows the comparison of the total radius of gyration R_g and R_g^2 across researchers. We find that researchers can be roughly grouped into two main classes: *exploiters*, whose R_g^2 value is comparable to R_g (points along the diagonal), and *explorers* whose R_g^2 is considerably smaller than total

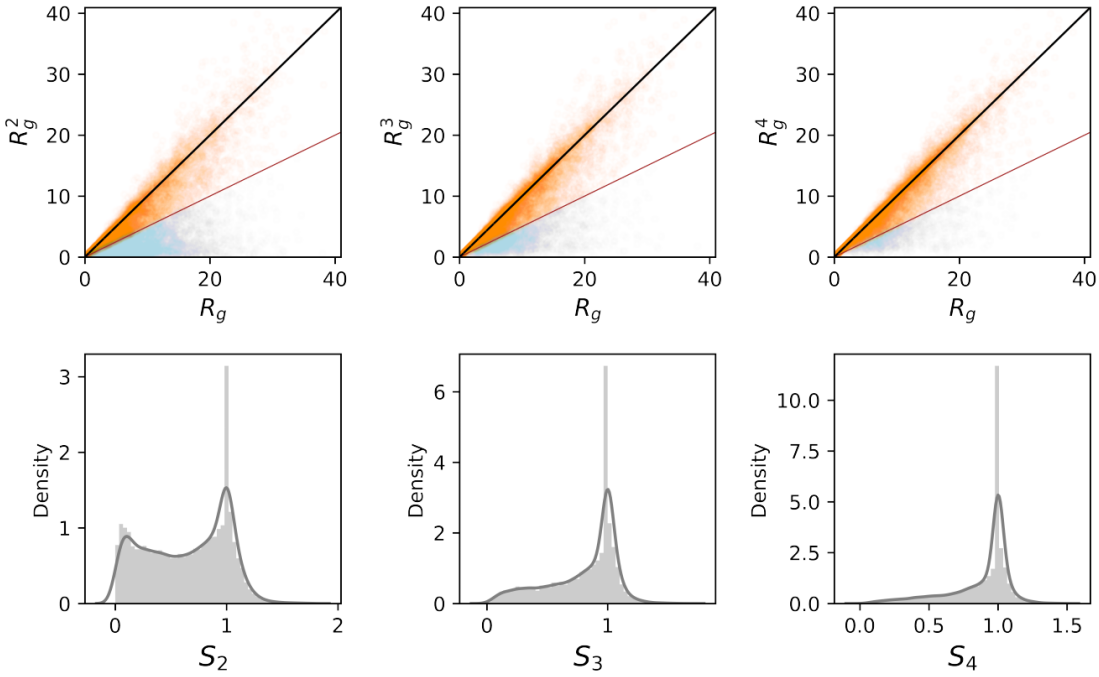


Figure 7.6: **Explorers and exploiters for larger values of k.** Top row: same as Figure 7.5a, for $k = 2, 3, 4$. The red line corresponds to R_g^k - $R_g/2 = 0$, with points above the line (orange) corresponding to exploiters, and points below the line (blue) corresponding to explorers. Bottom row: same as Figure 7.5b, for $k = 2, 3, 4$. The bimodality observed for $k = 2$ vanishes for larger values of k.

R_g (points closer to the x-axis). The two classes are more evident when considering the distribution of $S_2 = R_g^2/R_g$, showing two peaks corresponding to the two populations of explorers and exploiters (Figure 7.5b). This bimodality disappears when considering larger values of k (Figure 7.6), supporting the use of $k = 2$ to distinguish the two classes.

By design, the difference between explorers and exploiters will affect the research space spanned by scientific trajectories over time. While the radius of gyration considers the gravitation of a researcher around a particular centre of attraction (the centre of gravity), other measures focus on the dynamics of departure from an original starting. In mobility analysis, this is typically quantified by the mean squared displacement (MSD) [Klafter and Sokolov, 2011], a quantity that tracks the average distance travelled from the starting location over time (see Methods). The particular interest in MSD stems from the fact that simple diffusion processes in homogeneous spaces observe a functional scaling with time, MSD$(t) \sim t^\beta$, with the exponent β indicating a super- or sub-diffusive process. In our case, we find that, while both classes make jumps of similar size and duration (Figure 7.7) and have a similar sublinear MSD growth, explorers span a larger fraction of the knowledge space early in their career, as indicated by a faster MSD growth between 5 and 15 years (Figure 7.5c). This difference decreases in the later phase of their career (around 20 years), indicating that researchers tend to explore mostly in the middle of their academic life, while senior scientists tend to exploit more their previous research. This finding comforts prior observations that scientists become less disruptive and more critical of emerging work as they age [Cui et al., 2022].

Lastly, beyond differences in mobility patterns, we ask whether there are other characteristics that distinguish exploiters and explorers. To answer this question, we perform

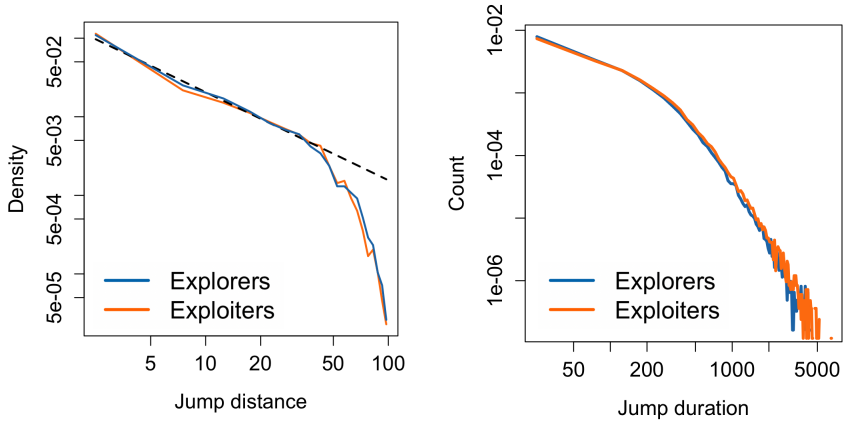


Figure 7.7: **Comparing jump distributions between explorers and exploiters.** The jump distance (left) and jump duration (right, in days) distributions are similar between explorers and exploiters (log-log plots).

a logistic regression to predict if an individual researcher is an explorer as a function of several attributes. To account for different trajectory lengths and field-specific behaviour, we control for the total number of articles published and the area of interest (Figure 7.1) in which the author has published the most. Figure 7.8 shows that, in line with the previous observation, explorers navigate broader regions of space, as measured by their maximum MSD achieved throughout their career, while exploiters tend to remain at the same location, measured by the proportion of their jumps being of distance 0.

Moreover, explorers cover more disciplines both within and across articles, and these disciplines tend to be cognitively distant, i.e. they are far in the field tag co-occurrence network [Singh et al., 2022]. When considering the association with specific developmental stages of scientific fields, we find that explorers publish in the earlier stages of the evolution of a field, a marker of pioneering activity and innovative work. Comforting this observation, we find a slightly higher disruptiveness for explorers ($p = 0.04$), a quantitative marker of innovative works quantifying the extent to which articles citing an article of interest also cite its sources (low disruptiveness) or not (high disruptiveness) [Park et al., 2023], measured here by the percentile of their most disruptive article. Finally, we observe that explorers and exploiters show similar impact, measured by the maximum citations obtained in one of their articles, and yearly productivity. We note however that results for the citation-based metrics are to be taken with care, as the citation network is only considering within-arXiv citations, and is therefore very incomplete [Clement et al., 2019] and subject to field-specific habits.

7.3 Discussion

In this study, we show that methods from mobility analysis applied to a low-dimensional representation of a knowledge space can help understand the scientific mobility of researchers. Using data from 1.5M articles from the pre-print repository arXiv across 30 years, we find that the mobility patterns of researchers resemble those found in human mobility studies. Flows between different regions of the knowledge space follow a gravity model, with an inverse relation to distance. This result is not an artefact from a

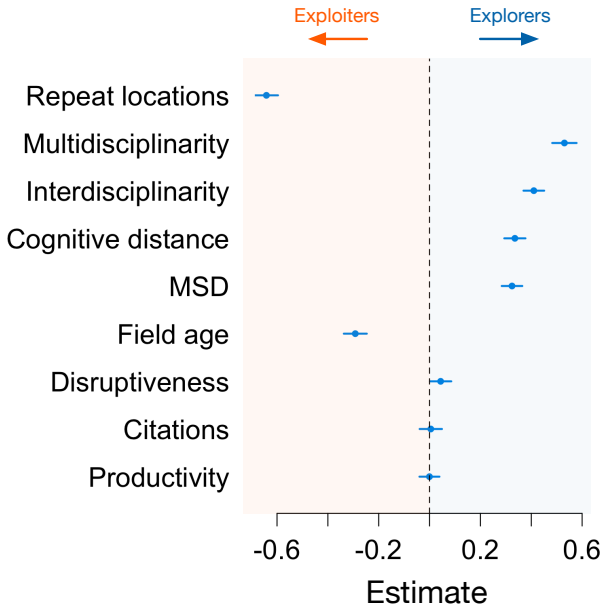


Figure 7.8: **Characteristics of explorers.** We compute a logistic regression of a binary variable y indicating whether an individual is an explorer $(y = 1)$ or an exploiter $(y = 0)$, for different characteristics of the researchers. For each attribute (further defined in Methods), we show the estimate and 95% confidence interval of the standardised coefficient of the regression, controlling for the number of articles and the main field of interest of the researcher. The differences between the two classes are all significant with p-values smaller than 10^{-5}, except for disruptiveness $(p = 0.04)$, citations and productivity which are non-significant $(p > 0.05)$. Repeat locations correspond to the proportion of jumps of size 0. Multidisciplinarity is the total number of unique field tags used across articles, and interdisciplinarity is the average number of field tags per article. Cognitive distance is the maximum disciplinary distance spanned by the researcher, and Field age is the minimum normalised age of the fields across articles published by a researcher, both quantities being defined by Singh et al. [2022]. MSD denotes the maximum mean square displacement achieved through the career of researchers. Disruptiveness is the maximum percentile of disruptive index achieved by researchers across their articles, when compared to the whole of arXiv. Citations correspond to the logarithm of the maximum number of citations received by the articles of the researcher. Finally productivity is the average yearly number of articles of the researcher.

particular representation, as it holds across various embedding parameters and methods (see Figures 7.12 and 7.13). Furthermore, the model accuracy outperforms empirical results from human mobility studies [Simini et al., 2021], showing that despite its simplicity, this model is a promising foundation for future work. In addition, by analysing individual trajectories, we find that researchers can be categorised into *exploiters*, whose trajectories are bound to a particular area of the knowledge space, and *explorers*, who jump across boundaries and pioneer novel fields. This dichotomy is reminiscent of the "essential tension" between tradition and innovation in scientific research, where the desire to explore new promising areas is counterbalanced by the need to capitalise on the work done in the past [Kuhn, 1979, Aleta et al., 2019]. Here we identify this tension by uncovering two types of knowledge mobility patterns through the bimodality observed in R_g^2/R_g.

When considering the properties of scientific trajectories in the knowledge space, we observe that the mobility patterns of explorers and exploiters show sub-diffusive regimes. Theoretically, when considering the mobility of an individual in a homogeneous space, such as the initial hypercube or a regular lattice, the MSD follows a linear regime if the second moment of the step size distribution and the first moment of the waiting-time probability distribution are finite [Klafter and Sokolov, 2011]. In our case, the observed deviation from a linear MSD (Figure 7.5) may be due to the heavy-tailed waiting time probability distribution of the two groups of researchers (Figure 7.7). Another possibility is that the multi-scale nature of the knowledge space, as a complex and evolving cognitive construct, may be responsible for this trapped-like behaviour. Further investigation is needed to determine which of these approaches is more suitable for explaining the observed non-linearity in MSD behaviour.

We assumed stable categories of exploiters and explorers using the full trajectory of researchers, yet there can be variation throughout their career. For example, we observed ageing patterns within trajectories, with MSD of explorers and exploiters showing similar values within the first 5 years after their first publication, after which MSD values for explorers are significantly larger (Figure 7.5c). This could indicate that explorers go through two phases: a first phase where they are staying within a few most visited locations, followed by an exploratory behaviour towards other locations. Such a behaviour could be formally captured using the concept of "intermittent behaviour" from stochastic processes [Lanoiselée and Grebenkov, 2017]. Future work could investigate such temporal patterns across research trajectories, for example by using time windows or the convex hull method to analyse dynamic profiles at a finer scale, and assess whether phases might correspond to institutional constraints, with some environment fostering the individual development towards more exploratory patterns.

Our framework relies on the method used to define the knowledge space. There is no ground truth in the use of embedding methods, and each can bias results towards specific idiosyncratic properties. However, we have shown that a variety of parametric (tSNE) and non-parametric (UMAP, PaCMap) methods yield qualitatively similar results, both in terms of long-tailed jump distribution (Figure 7.12) and gravity model fit (Figure 7.13). This indicates that, despite some variations coming from the structure of the space itself, the general mobility patterns uncovered here are not space representation artefacts.

Our work is focused on a dataset of arXiv pre-prints. This dataset provides a high precision for the identification of subfields, which is useful for both the construction of the knowledge space and the computation of features such as field age [Singh et al., 2022]. Yet, it does not contain all the research articles or fields and its citation network is incomplete. It is therefore yet unclear how our findings generalise to other disciplines, for

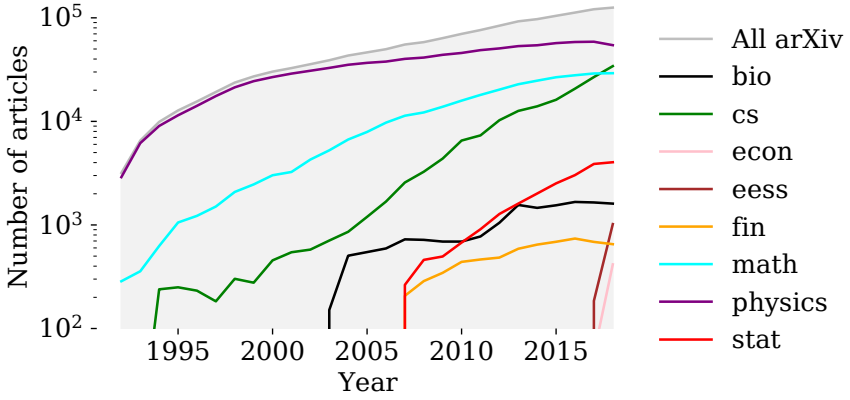


Figure 7.9: **Growth in the number of articles submitted to the major fields of arXiv between 1992 and 2018**. We show the cumulative number of articles over time for the 9 major fields in arXiv, totalling $1,456,403$ articles. Fields consist of: Quantitative Biology (bio), Computer Science (cs), Economy (econ), Electrical Engineering and Systems Science (eess), Quantitative Finance (fin), Mathematics (math), Physics (physics), and Statistics (stat). We use a log-scaling for an easier visualisation of the more recent fields.

example when considering the humanities or social sciences. Future work should explore the reproducibility of our findings across larger and diverse datasets, leveraging other field identification methods, such as the ones using Natural Language Processing [Beltagy et al., 2019].

While our study is focused on the description of individual trajectories, most articles are team-authored [Wuchty et al., 2007], and chaperoning patterns are fundamental to scientific careers [Sekara et al., 2018]. Therefore, future works could study the coupling between individual trajectories, leading to correlated patterns and ultimately collective flows. In addition, the gravity model could be extended to incorporate variables corresponding to local attributes, such as impact (e.g. through citations), field age, funding, etc. These features might act as biasing forces shaping collective flows towards certain areas of the knowledge space. On a macroscopic level, these fields can affect mobility, in the same way that force fields affect the trajectories of particles in physics. Novel methods based on deep learning, such as a Deep Gravity Model [Simini et al., 2021], coupled with more extensive data on citations and funding, could help extend our work.

Overall, the insights gained from leveraging a mobility analysis in the knowledge space could help study the effect of policies on knowledge exploration and exploitation, with applications for funding agencies and more generally the evaluation of research.

7.4 Methods

7.4.1 Overview of the arXiv dataset

In our study we use a previously published dataset consisting of article metadata from the arXiv preprint repository [Singh et al., 2022]. The dataset consists of $1,456,403$ articles published between 1992 and 2018, covering mainly the fields of physics, mathematics and computer science, and to a lesser extent Quantitative Biology, Statistics, Finance, Economy, and Engineering. We note in particular the important rise of Computer Science articles in the past decade, with a published rate bound to soon outweigh the physics field, which was at the core of the early arXiv usage (Figure 7.9).

When uploading an article, the submitting author selects a main, primary tag identifying the core discipline, along with secondary tags if needed. In most cases, arXiv require authors who are submitting papers to a subject category for the first time to get an endorsement from an established arXiv author, as a quality control mechanism. The tags span 175 predefined subfields, such as Quantum Algebra (math.QA) or Signal Processing (eess.SP), all indicated on the website's main page. These subfields have remained relatively stable in time [Singh et al., 2022]. Moreover, there is a strong incentive for authors to select appropriate fields, as arXiv proposes a subscription service to a daily digest email system to automatically receive novel submitted articles containing a specific field tag. As such, the tag system is directly tied to a relevant audience for the publishing individual, which incentivises for an accurate self-report.

7.4.2 Low dimensional embeddings

To reduce the dimensionality of the initial 175-dimensional field space, we use the tSNE algorithm, an unsupervised, parametric dimensionality reduction technique that retains the local data structure in the latent space [Van der Maaten and Hinton, 2008, Van der Maaten, 2009]. The tSNE method captures much of the local structure of the high-dimensional data, while also revealing global structure such as the presence of clusters at different scales. The visualisation of the resulting embedding of the arXiv knowledge space into a two dimensional space is shown in Figure 7.1. Each point corresponds to one of the $49,575$ observed combinations of field tags within arXiv articles. We note that permutations of tags map to the same point, so that our analysis does not depend on the order of tags.

For the implementation of the tSNE algorithm we use the scikit-learn package [Pedregosa et al., 2011]. The dimension of the embedded space is set to 2. The main parameters of the embedding method, such as learning rate, number of iterations and early exaggeration parameters are set to default values. In order to test the robustness of the tSNE embedding to varying parameters, we generated tSNE mapping for different perplexity levels p (signifying the nearest neighbours) and learning rate parameters LR of the algorithm, and plotted the pairwise distance distribution between randomly sampled points across different settings (Figure 7.10). We find remarkable stability across various parameters of the tSNE suggested in [Wang et al., 2021], including perplexity levels, indicating that choosing different tSNE parameters would not strongly affect the results.

7.4.3 Fitting procedure for the gravity model

In order to fit the gravity model, we used a linear regression of log-transformed variables. For each resolution level, we first computed for each year starting in 1997 the number of jumps between a source cell i and a target cell j (with $i \neq j$) and the number of articles published in each cell in the 5 preceding years. In order to account for low sample size, we used a pseudo-count of 1 added to raw visit counts: $V_i \leftarrow V_i + 1$. We then computed the natural logarithm of all quantities, and used these log-transformed values for the regression analysis. Since there is a much larger number of small flow values compare with large flow values (Figure 7.4), we used a binning technique to avoid over-fitting our model to low flow values. For this, we cut the obtained log-flow values into 100 bins containing an equal number of points, and merged bins with the same breakpoints, resulting in 42 final bins. We then averaged the log-transformed features (flow, visits, distance) within these bins, and used these average values to fit the gravity model, using the lm

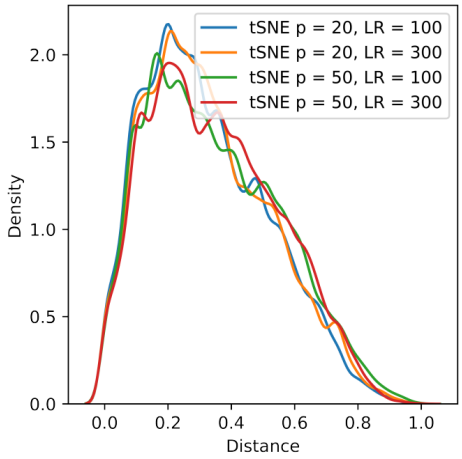


Figure 7.10: **Comparison of pairwise distance distributions across various tSNE parameters.** Density distributions of the distance between randomly sampled pairs of points in different tSNE embeddings, with parameters shown in legend.

function from R 4.2.2. Figure 7.11 shows that the residuals of the model correspond to a normally distributed noise.

7.4.4 Radius of gyration

The radius of gyration measures the typical size of the territory spanned by the trajectory of an individual. To compute it, we first define the centre of mass R_{cm} of the trajectory across locations $i = 0, 1, \ldots, n$:

$$R_{cm} = \frac{\sum_{i=1}^{n} M_i R_i}{\sum_{i=1}^{n} M_i}, \tag{7.2}$$

where M_i is the frequency of visitation of each location i, i.e. the number of times location i is visited by the individual, and R_i is the radius vector characterising the location in the knowledge space with respect to the chosen centre of coordinates. The radius of gyration is then defined as the characteristic distance from the centre of mass:

$$R_g = \sqrt{\frac{\sum_{i=1}^{n} M_i (R_i - R_{cm})^2}{\sum_{i=1}^{n} M_i}}. \tag{7.3}$$

In order to estimate the influence of a few locations over the trajectory, we define the k-th radius of gyration by considering only the top k most visited location:

$$R_g^k = \sqrt{\frac{\sum_{i=1}^{k} M_i (R_i - R_{cm}^k)^2}{\sum_{i=1}^{k} M_i}}, \tag{7.4}$$

where R_{cm}^k is the centre of mass using the top k most visited locations.

7.4.5 Mean squared displacement

The mean squared displacement (MSD) at time t for a trajectory is defined as the deviation of the position of a walker (in our case, a researcher) with respect to a reference position over time:

$$\text{MSD}(t) = \langle |x(t) - x(0)|^2 \rangle \tag{7.5}$$

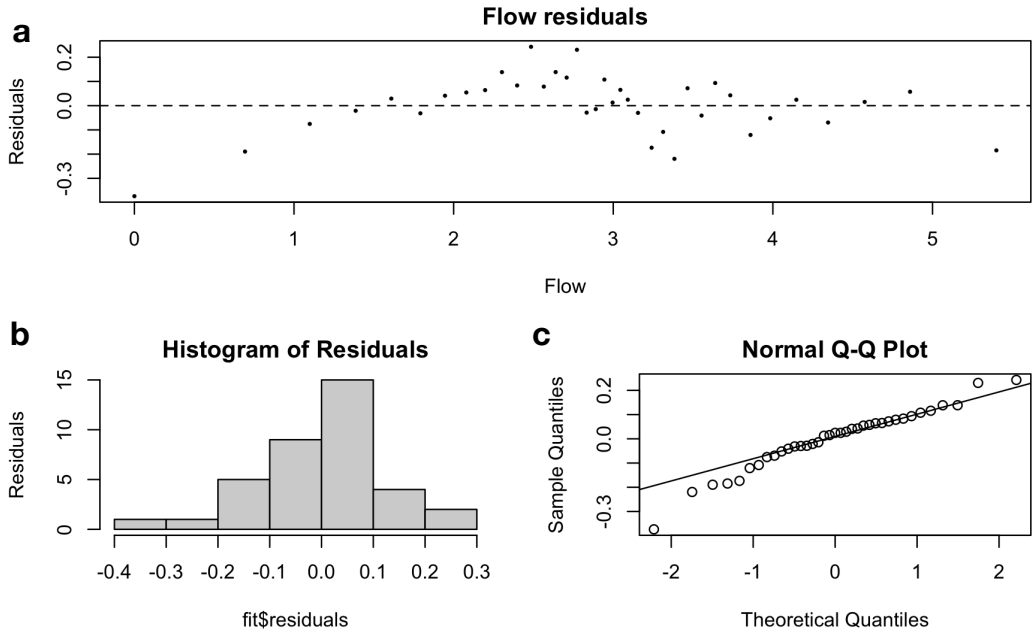


Figure 7.11: **Residual plots for the gravity model.** We test that the assumption of normality of residuals hold in our regression analysis. **a.** Value of residuals as a function of the fitted (log) flow values. We find that very low flow values are slightly over-estimated, which might stem from the pseudo-counting method used. **b.** Histogram of the residuals. **c.** The residuals are normally distributed, as can be assessed using a Q-Q plot.

where $x(t)$ stands for the position of researcher at time t since the first article, and $x(0)$ stands for the starting point of the trajectory.

7.4.6 Logistic regression for explorers vs. exploiters

In order to explore the characteristics associated with explorers in Figure 7.8, we compute a logistic regression with dependent variable y_i, a binary variable indicating whether an individual i is an explorer ($y_i = 1$) or an exploiter ($y_i = 0$), and independent variables for various individual features x_i. We control for the main field F_i in which the author has published (given by the most represented field tag across their articles), as well as the number of articles N_i of the researcher. The fields were encoded as factors. We used the glm function in R to fit the model $y_i \sim x_i + N_i + F_i$, with parameter $\mathrm{family} = \mathrm{binomial}$ set to a logistic regression. Regression summaries were obtained using the summ function from the jtools package in R, with parameters $\mathrm{scale}{=}\mathrm{T}$ to standardise the regression coefficients by scaling and mean-centring input data, and $\mathrm{confint}{=}\mathrm{T}$ to obtain 95% confidence intervals.

7.4.7 Innovation, disruptiveness and impact

To measure the innovative level of a work, we used two methods. First, we computed for each article how early it occurs within the fields that it mentions. To do so, we computed the minimum *rescaled time* (RT) across its associated field tags, using the method described by Singh et al. [2022]. The rescaled time is a normalised quantity that allows us to associate an article to a developmental stage of a field (early, peak, or late phase) even when fields have drastically different rise and fall durations. We then computed for each

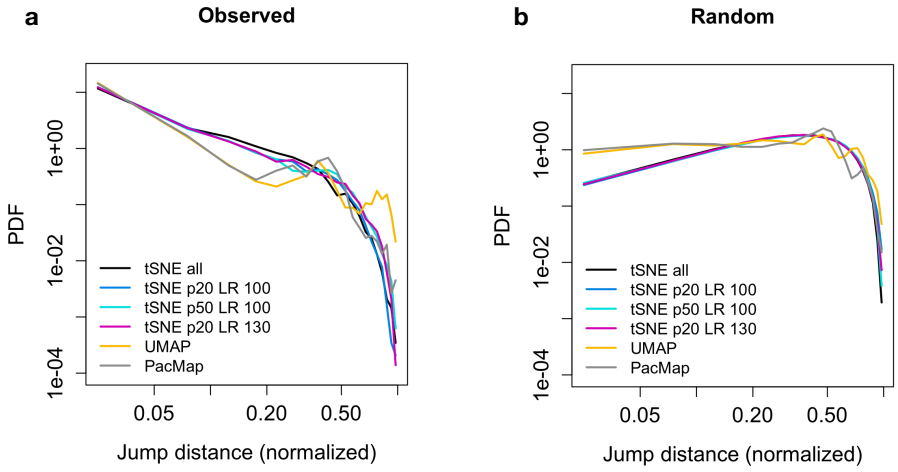


Figure 7.12: **Comparison of jump distributions across embeddings methods. a.** We show the variation of the jump distribution when using different parameters for the tSNE embedding (p is perplexity, and LR is Learning Rate) as well as other embedding methods (PaCMap and UMAP). In order to compare between the different embeddings, the Jump distance is normalised by the maximum distance for each embedding. **b.** For each author, we select random locations across all accessible points in the embedding (i.e. unique existing locations in the dataset) and plot the corresponding randomised jump distributions across embeddings.

researcher the minimum RT value achieved across their articles, defining the "Field age".

Second, we used another independent method to assess the innovative potential of the articles. This method evaluates how disruptive an article is by comparing the attention it receives to the articles it cites. Citation data was obtained from Clement et al. [2019]. The disruptive index (DI) was then computed using the method of Wu et al. [2019] for each article. For each author, we computed the maximum DI across their articles. Finally, we computed the percentile of the obtained value across articles to compute the disruptiveness of an author.

Finally, for each author i, we computed the maximum number of citations $\hat{c}_i$ received by any of their articles across their career. Since citation counts are distributed with a heavy-tailed function, we used the transformation $\log(\hat{c}_i + 1)$ to quantify the impact.

7.4.8 Cognitive Distance

We observe in Figure 7.8 that compared to exploiters, explorers use a larger number of field tags per article, as well as a larger number of unique tags across their articles. However some tags might be more closely related than others in terms of research area, which the simple measure for linear estimate of tags used does not differentiate. To account for this effect, we use the network based cognitive distance measure from Singh et al. [2022], where the cognitive distance C_{ij} between field tags i and j is the weighted distance along the shortest path between tags i and j in the tag co-occurrence network.

7.4.9 Robustness with respect to the embedding method

In order to assess the robustness of our results, we tested the impact of different embedding methods, parameters, as well as subsamples of the data on the jump distribution (Figure 7.12). Beyond tSNE, we evaluated the robustness of our analysis using PaCMAP and UMAP embeddings.

The Uniform Manifold Approximation and Projection (UMAP) [McInnes et al., 2018] has its theoretical foundations in manifold theory and topological data analysis. At a macroscopic level, UMAP uses local manifold approximations and fuzzy simplicial sets to construct topological representations of data in high and low dimensions. It then minimises the cross-entropy between the two topological representations to find an optimal lower-dimensional representation. UMAP can also be understood as a k-neighbour based graph learning algorithm that finds the best representation of weighted graphs in lower dimensions.

The Pairwise Controlled Manifold Approximation (PaCMAP) algorithm [Wang et al., 2021] is also a graph-based technique that identifies three sets of pairs, namely neighbour pairs, mid-near pairs and further pairs. It then systematically optimises its loss function using a custom gradient descent algorithm to find a lower dimensional representation that preserves both local and global structures.

We find that the jump distance distributions shows a similar long-tail decay for both methods (Figure 7.12a). In addition, we find that the gravity model has stable results in UMAP and PaCMAP (Figure 7.13) contexts, though we find an overall smaller exponent for the distance, closer to $\gamma \simeq 0.5$.

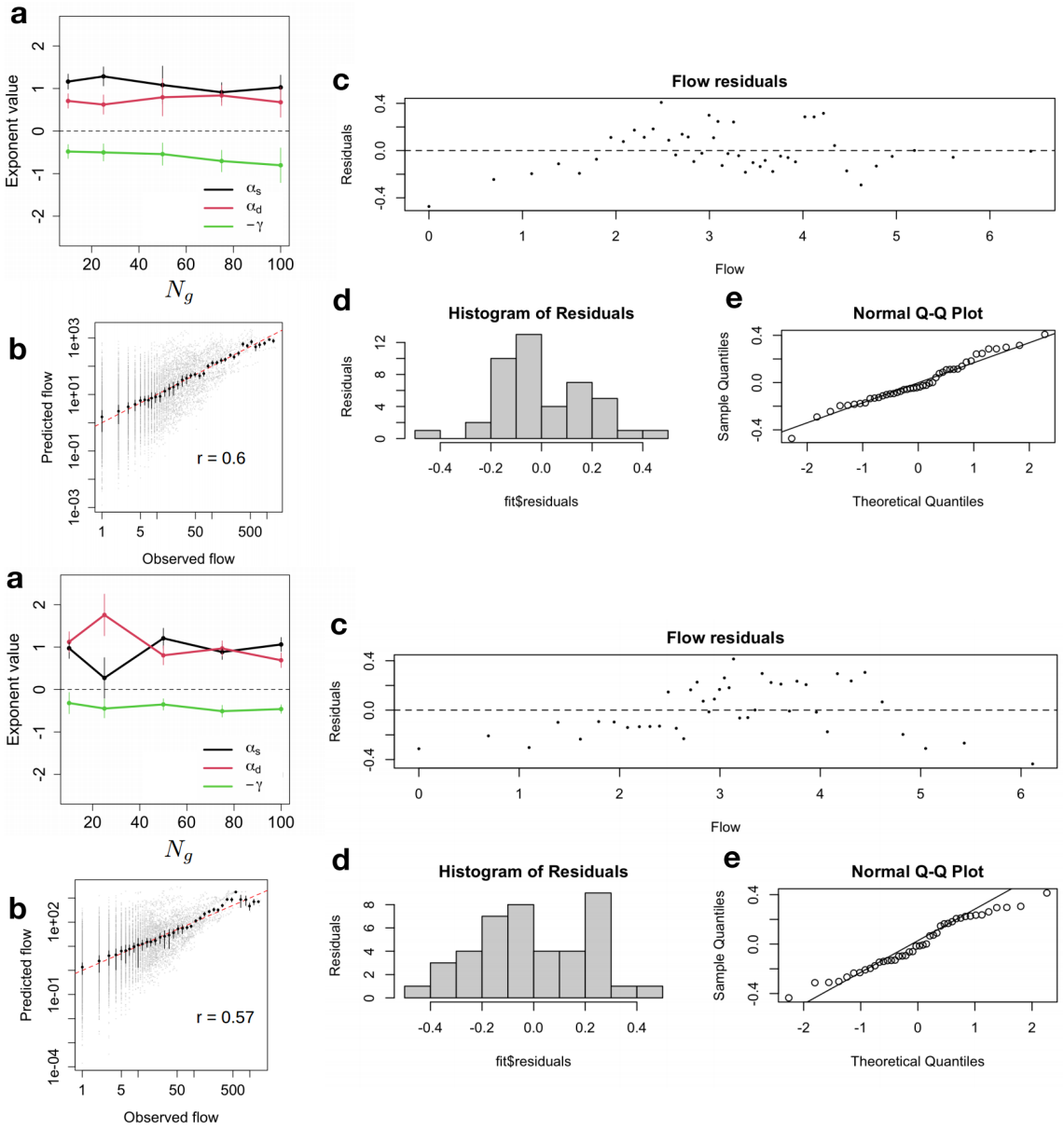


Figure 7.13: **Results of the fit of the gravity model for embeddings UMAP (top) and PaCMAP (bottom). a-b.** Same as Figure 7.4b-c. **c-e.** Same as Figure 7.11.

Conclusion

Summary

This thesis aimed at taking advantage of node orderings to improve algorithms in practice on large real-world networks. Chapter 1 introduced the topic for a general audience and Chapter 2 exposed the notations and concepts involved in the subsequent chapters. In particular, we presented some key ideas of graph theory and algorithmic complexity, as well as the specific challenges of real-world networks and effective time measurements.

In Chapter 3, we reviewed existing works of the literature involving node orderings. We proposed a classification with three aspects. First, orderings are created with a certain mechanism in mind, such as giving close indices to nodes that are close in the graph. Second, orderings are obtained with an algorithm that follows one of few patterns, such as optimising an objective function or reducing the graph to obtain a more manageable subgraph. Third, orderings arise in various application domains of graph algorithmics, like compression, mining or robustness, to name a few. We also showed a reformulation with orderings for some theoretical graph problems. Finally, we discussed the overhead issue of orderings, which implies to decide on the balance between the cost of finding orderings and the benefits that they yield.

Chapter 4 illustrated a first way in which orderings can improve algorithms in practice: they can reduce the rate of cache-misses. A cache-miss happens when the data required by an algorithm is not stored in the cache, which requires to fetch it further in the memory and incurs an additional waiting time. Orderings are able to mitigate this issue by bringing together in memory the nodes that graph algorithms will likely need at the same. One major interest of this method is that it accelerates operations that are key to most graph algorithms, thus benefiting them regardless of their precise objective or implementation. We replicated a paper that proposes an objective function and a practical ordering algorithm for the cache optimisation problem, and confirmed that it leads to a significant acceleration for a selection of standard graph algorithms. However, we found that some simpler and more scalable methods are competitive, highlighting the previously mentioned issue of overhead.

A second way of improving algorithms with orderings was shown in Chapter 5, where we saw that the number of operations in triangle listing algorithms explicitly depends on the ordering. We proved the hardness of finding optimal orderings, and gave additional results for approximation and relaxation. Given these proofs and the objective of scaling to large real-world networks, we proposed three heuristics to obtain orderings that are efficient although not optimal. They offer a range of options depending on the tolerance for overhead: one favours ordering quality, another favours ordering time, and the last one is an intermediate option that we set as default in our open-source implementation. Experiments showed that our set of heuristics outperforms existing methods in different

settings. Overall, our results pointed at the power of orderings to address a specific algorithm to make it faster or more scalable by reducing its number of operations.

In Chapter 6, we gave a last example of the value of orderings for improving algorithms. We focused on the minimum vertex cover problem, which is notoriously difficult in theory but has efficient practical heuristics for large graphs. We proposed a simple and effective method using greedy algorithms with appropriate priority functions that correspond to a node ordering. To certify the quality of a heuristic result, we obtained instance-specific lower-bounds and measured how close the heuristic result is to the bound. Through extensive experiments, we demonstrated that the certified quality was excellent for all the networks that we examined. This underlines both the efficacy of existing heuristics for vertex cover and the relevance of the certification method using these lower-bounds.

Chapter 7 left the scope of node orderings to focus on the network science question of scientific mobility. We processed a dataset of papers tagged with dozens of different scientific branches and fields to create a two-dimensional space representing knowledge. On this space, we analysed the trajectory of individual researchers throughout their career, and found that their collective movement is consistent with a gravity model, where the number of papers in a field is the driving force, and the cognitive distance causes a decay. We also reported a split in the population between exploiters, who deepen the knowledge in a small area of the space, and explorers who travel long distances for interdisciplinary research.

Perspectives

The contributions of this thesis and the vast anterior literature give evidence that node orderings are a crucial element of graph algorithms. For this reason, we believe that a more systematic analysis of orderings could lead to new results. First of all, when we suspect that an algorithm or a technique is impacted by node orderings, we could test existing orderings systematically to gain insight on which mechanisms work best. The classification of Chapter 3 could contribute to this purpose, as well as the various orderings that our repositories implement (see Introduction). Second, orderings defined by objective functions could benefit from a deeper theoretical analysis: a classical way to handle them, which we followed in Chapter 5, is to prove their hardness, then to design heuristics that are intuitively believed to improve the value of the function. While experimental results are a valid way to give evidence of the success of such heuristics, a stronger mathematical grounding of the initial hard problem is also important. In this direction, it is interesting to design approximation algorithms or schemes, and to find classes of graphs or parameters for which the problem is tractable. Finding bounds for the optimisation is also an interesting track to explore as in Chapter 6: bounds reveal how far heuristics are from optimal, which indicates whether stronger efforts to approach the optimum are worthwhile.

In some graph problems, the edge ordering may be more relevant than the node ordering. Note that there are natural ways to define one from the other: given a node ordering, one can order the edges according to the smaller index of their nodes, breaking ties with the second node; conversely, given an edge ordering, one can order the nodes by the index of the first edge in which they appear. In this thesis, we focused on node orderings based on two assumptions: that the graph is stored as adjacency lists, and that the main atomic operation of algorithms is listing the neighbours of a node. However, edge orderings prove relevant in other contexts, when edges are the key element instead of nodes.

The first assumption is invalidated when the graph is stored as a list of edges, which is a standard format to share datasets in a file. The second assumption is incorrect for some edge-based algorithms, among which the greedy matching algorithm described in Chapter 6. Likewise, the power iteration method for Pagerank and eigenvector centralities rely on edge enumeration, and the edge ordering may impact their convergence rate. In an adjacency matrix, each element corresponds to an edge; enumerating the edges following their corresponding node ordering boils down to reading the matrix row after row. Yet, this is just one of many options to enumerate the edges, and other orderings have been proposed. Among them is the ordering following two-dimensional space-filling curves: drawing a curve that crosses all the elements of the adjacency matrix provides an ordering of the edges that does not correspond to a node ordering. Research on edge orderings could help us understand how node and edge orderings relate, and which type of ordering is best suited for a particular graph problem.

Orderings can be seen as a hidden way to make algorithms faster or stronger and, as such, could be integrated in graph processing systems. Such software systems typically allow users to store, update and query large graphs efficiently in a parallel or distributed manner. To do so, they need specialised algorithms and data structures to perform operations such as graph traversals, pattern mining or community detection. Ordering the nodes in a specific way could help accelerating these tasks in general, as seen with the cache optimisation in Chapter 4. It is even possible to maintain several orderings in a transparent manner, so that each type of query is ready to be processed with an appropriate ordering. For example, if the system maintains an ordering of nodes based on their degree, it can process more efficiently the queries that rely on degree, like triangle listing in Chapter 5 or other pattern mining tasks. Simultaneously, the system can maintain a node ordering with locality mechanism to improve the results of algorithms that benefit from this property, such as graph partitioning. Given the limited number of ordering mechanisms involved across application domains, graph processing systems could face a diversity of queries with only a few orderings to store, making the time and space overhead reasonable.

Furthermore, there are opportunities to apply ordering techniques to more sophisticated graph models, including temporal and dynamic graphs. For a graph that evolves over time due to node and edge additions or deletions, the challenge is to maintain an accurate or approximate ordering with specific properties. While maintaining the degree ordering is feasible with a priority queue, it is not straightforward for more elaborate orderings such as the core ordering or Slashburn. Indeed, they are based on a decomposition of the graph that can change dramatically by the addition of a single edge. An extension to the maintenance problem could consist in designing orderings that are robust to graph modifications. Such orderings would be top candidates to address the overhead issue: one can accept to spend more time in computing an ordering if there is a guarantee that it can be updated in little time when the graph changes.

Finally, orderings are not limited to graph problems and can also relate to the algorithmic problem of embedding. Embedding can be seen as a multi-dimensional centrality that seeks an optimal placement of the elements to satisfy certain constraints, such as clustering elements or keeping the variability on few dimensions. By exploring the use of orderings in this non-graph problem, we may discover new insights and techniques that transfer to graph problems. For example, the concept of locality in node orderings can be derived from a node embedding, as hinted in Chapter 7. In this way, orderings can serve as a powerful tool to address a broader spectrum of algorithmic problems.

Bibliography

Amit Agarwal, Moses Charikar, Konstantin Makarychev, and Yury Makarychev. O(log n) approximation algorithms for min UnCut, min 2CNF deletion, and directed cut problems. STOC, 2005. URL https://doi.org/10.1145/1060590.1060675.

Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a modern processor: where does time go? VLDB, 1999.

Eleni C Akrida. Temporal vertex cover with a sliding time window. *Journal of Computer and System Sciences*, 2020.

Mohammad Al Hasan and Vachik S. Dave. Triangle counting in large networks: a review. *WIREs Data Mining Knowl Discov*, 2018. doi: 10.1002/widm.1226. URL https://onlinelibrary.wiley.com/doi/10.1002/widm.1226.

Reka Albert, Hawoong Jeong, and Albert-László Barabási. Error and attack tolerance of complex networks. *Nature*, 2000. doi: 10.1038/35019019. URL http://arxiv.org/abs/cond-mat/0008064.

Laura Alessandretti, Ulf Aslak, and Sune Lehmann. The scales of human mobility. *Nature*, 2020.

Alberto Aleta, Sandro Meloni, Nicola Perra, and Yamir Moreno. Explore with caution: mapping the evolution of scientific interest in physics. *EPJ Data Science*, 2019. URL https://epjdatascience.springeropen.com/articles/10.1140/epjds/s13688-019-0205-9.

Reid Andersen and Fan Chung. Detecting Sharp Drops in PageRank and a Simplified Local Partitioning Algorithm. In *Theory and Applications of Models of Computation*, 2007. doi: 10.1007/978-3-540-72504-6_1.

Eric Angel, Romain Campigotto, and Christian Laforest. Implementation and Comparison of Heuristics for the Vertex Cover Problem on Huge Graphs. In *Experimental Algorithms*. 2012. doi: 10.1007/978-3-642-30850-5_5. URL http://link.springer.com/10.1007/978-3-642-30850-5_5.

Alberto Apostolico and Guido Drovandi. Graph Compression by BFS. *Algorithms*, 2009. doi: 10.3390/a2031031. URL http://www.mdpi.com/1999-4893/2/3/1031.

Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. Rabbit Order: Just-in-Time Parallel Reordering for Fast Graph Analysis. In *IPDPS*, 2016. doi: 10.1109/IPDPS.2016.110. URL http://ieeexplore.ieee.org/document/7515998/.

Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. Fast Parallel Algorithms for Counting and Listing Triangles in Big Graphs. *TKDD*, 2019. URL https://doi.org/10.1145/3365676.

Yuichi Asahiro, Kazuo Iwama, Hisao Tamaki, and Takeshi Tokuyama. Greedily Finding a Dense Subgraph. *Journal of Algorithms*, 2000. doi: 10.1006/jagm.1999.1062. URL https:

//www.sciencedirect.com/science/article/pii/S0196677499910623.

Eyjolfur Asgeirsson and Cliff Stein. Vertex Cover Approximations on Random Graphs. In *Experimental Algorithms*. 2007. doi: 10.1007/978-3-540-72845-0_22. URL http://link.springer.com/10.1007/978-3-540-72845-0_22.

Arda Asik, Ibrahim Bugra Demir, Berker Demirel, Baris Batuhan Topal, and Kamer Kaya. Vertex Ordering Algorithms for Graph Coloring Problem. *arXiv*, 2020. URL http://arxiv.org/abs/2008.11454.

Sepehr Assadi, Krzysztof Onak, Baruch Schieber, and Shay Solomon. Fully dynamic maximal independent set with sublinear update time. STOC 2018, 2018. URL https://doi.org/10.1145/3188745.3188922.

David Avis and Tomokazu Imamura. A list heuristic for vertex cover. *Operations Research Letters*, 2007. doi: 10.1016/j.orl.2006.03.014. URL https://www.sciencedirect.com/science/article/pii/S0167637706000563.

Vignesh Balaji and Brandon Lucia. When is Graph Reordering an Optimization? In *IISWC*, 2018. doi: 10.1109/IISWC.2018.8573478. URL https://brandonlucia.com/pubs/iiswc18-lws.pdf.

Albert-László Barabási and Reka Albert. Emergence of scaling in random networks. *Science*, 1999. doi: 10.1126/science.286.5439.509. URL http://arxiv.org/abs/cond-mat/9910332.

Hugo Barbosa, Marc Barthelemy, Gourab Ghoshal, Charlotte R. James, Maxime Lenormand, Thomas Louail, Ronaldo Menezes, José J. Ramasco, Filippo Simini, and Marcello Tomasini. Human mobility: Models and applications. *Physics Reports*, 2018. doi: 10.1016/j.physrep.2018.01.001. URL https://www.sciencedirect.com/science/article/pii/S037015731830022X.

Hugo S. Barbosa, Fernando B. de Lima Neto, Alexandre Evsukoff, and Ronaldo Menezes. Returners and Explorers Dichotomy in Web Browsing Behavior—A Human Mobility Approach. In *CompleNet*. 2016. URL https://doi.org/10.1007/978-3-319-30569-1_13.

Reet Barik, Marco Minutoli, Mahantesh Halappanavar, Nathan R. Tallent, and Ananth Kalyanaraman. Vertex Reordering for Real-World Graphs and Applications. In *IISWC*, 2020. URL https://eecs.wsu.edu/~ananth/papers/Barik_IISWC20.pdf.

Vladimir Batagelj and Matjaž Zaveršnik. Generalized Cores. Technical report, arXiv, 2002. URL http://arxiv.org/abs/cs/0202039.

Vladimir Batagelj and Matjaž Zaveršnik. An O(m) Algorithm for Cores Decomposition of Networks. *arXiv*, 2003. URL http://arxiv.org/abs/cs/0310049.

Federico Battiston, Federico Musciotto, Dashun Wang, Albert-László Barabási, Michael Szell, and Roberta Sinatra. Taking census of physics. *Nature Reviews Physics*, 2019.

Alex Bavelas. Communication Patterns in Task-Oriented Groups. 1950. URL https://hal.science/hal-03266728.

Abbas Bazzi, Samuel Fiorini, Sebastian Pokutta, and Ola Svensson. No Small Linear Program Approximates Vertex Cover Within a Factor 2-epsilon. *Mathematics of OR*, 2018. doi: 10.1287/moor.2017.0918. URL http://pubsonline.informs.org/doi/10.1287/moor.2017.0918.

Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. KDD, 2008. URL https://doi.org/10.1145/1401890.1401898.

Alexander V Belikov, Andrey Rzhetsky, and James Evans. Prediction of robust scientific

facts from literature. *Nature Machine Intelligence*, 2022.

Iz Beltagy, Kyle Lo, and Arman Cohan. SciBERT: A Pretrained Language Model for Scientific Text. In *EMNLP-IJCNLP*, 2019. doi: 10.18653/v1/D19-1371. URL https://aclanthology.org/D19-1371.

Maciej Besta and Torsten Hoefler. Survey and Taxonomy of Lossless Graph Compression and Space-Efficient Graph Representations. *arXiv*, 2019. URL http://arxiv.org/abs/1806.01799.

Therese Biedl, Timothy Chan, Yashar Ganjali, Mohammad Taghi Hajiaghayi, and David R. Wood. Balanced vertex-orderings of graphs. *Discrete Applied Mathematics*, 2005. URL https://linkinghub.elsevier.com/retrieve/pii/S0166218X04003828.

Chris Biemann, Lachezar Krumov, Stefanie Roos, and Karsten Weihe. Network Motifs Are a Powerful Tool for Semantic Distinction. In *Towards a Theoretical Framework for Analyzing Complex Linguistic Networks*. 2016. doi: 10.1007/978-3-662-47238-5_4. URL https://www.researchgate.net/publication/282069627_Network_Motifs_Are_a_Powerful_Tool_for_Semantic_Distinction.

Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *J. Stat. Mech.*, 2008. doi: 10.1088/1742-5468/2008/10/P10008. URL http://arxiv.org/abs/0803.0476.

Norbert Blum. A new approach to maximum matching in general graphs. In *Automata, Languages and Programming*, 1990. doi: 10.1007/BFb0032060.

Thomas Bläsius, Philipp Fischbeck, Tobias Friedrich, and Maximilian Katzmann. Solving Vertex Cover in Polynomial Time on Hyperbolic Random Graphs. Technical report, arXiv, 2020. URL http://arxiv.org/abs/1904.12503.

Hans L. Bodlaender, Fedor V. Fomin, Arie M. C. A. Koster, Dieter Kratsch, and Dimitrios M. Thilikos. A Note on Exact Algorithms for Vertex Ordering Problems on Graphs. *Theory Comput Syst*, 2012. doi: 10.1007/s00224-011-9312-0. URL http://link.springer.com/10.1007/s00224-011-9312-0.

Paolo Boldi and Sebastiano Vigna. The webgraph framework I: compression techniques. In *WWW*, 2004. doi: 10.1145/988672.988752. URL http://portal.acm.org/citation.cfm?doid=988672.988752. https://law.di.unimi.it/datasets.php.

Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. Technical report, arXiv, 2011. URL http://arxiv.org/abs/1011.5425.

Eric Bonabeau, Marco Dorigo, Guy Theraulaz, and Guy Theraulaz. Swarm intelligence: from natural to artificial systems. 1999.

Lutz Bornmann, Robin Haunschild, and Rüdiger Mutz. Growth rates of modern science: a latent piecewise growth curve approach to model publication numbers from established and new literature databases. *Humanities and Social Sciences Communications*, 2021.

Endre Boros and Vladimir Gurvich. On complexity of algorithms for modeling disease transmission and optimal vaccination strategy. 2007.

Florian Bourse, Marc Lelarge, and Milan Vojnovic. Balanced graph edge partition. In *KDD*, 2014. doi: 10.1145/2623330.2623660. URL https://dl.acm.org/doi/10.1145/2623330.2623660.

Paweł Brach, Marek Cygan, Jakub Łącki, and Piotr Sankowski. Algorithmic Complexity of Power Law Networks. In *Symposium on Discrete Algorithms*, 2016. doi: 10.1137/1.

9781611974331.ch91. URL http://epubs.siam.org/doi/10.1137/1.9781611974331.ch91.

Marco Bressan. Efficient and near-optimal algorithms for sampling connected subgraphs. In *STOC*, 2021. URL https://doi.org/10.1145/3406325.3451042.

Marco Bressan and Marc Roth. Exact and approximate pattern counting in degenerate graphs: New algorithms, hardness results, and complexity dichotomies. In *FOCS*, 2021. URL http://arxiv.org/abs/2103.05588.

Marco Bressan, Enoch Peserico, and Luca Pretto. Sublinear algorithms for local graph centrality estimation. In *FOCS*, 2018. URL http://arxiv.org/abs/1404.1864.

Shaowei Cai, Jinkun Lin, and Chuan Luo. Finding A Small Vertex Cover in Massive Sparse Graphs: Construct, Local Search, and Preprocess. *JAIR*, 2017. doi: 10.1613/jair.5443. URL https://www.jair.org/index.php/jair/article/view/11071. Code provided at http://lcs.ios.ac.cn/ ~caisw/VC.html.

Shaowei Cai, Wenying Hou, Jinkun Lin, and Yuanjie Li. Improving Local Search for Minimum Weight Vertex Cover by Dynamic Strategies. In *IJCAI*, 2018. doi: 10.24963/ijcai.2018/196. URL https://www.ijcai.org/proceedings/2018/196.

Ümit Çatalyürek, Karen Devine, Marcelo Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Dorothea Wagner. More Recent Advances in (Hyper)Graph Partitioning. *ACM Computing Surveys*, 2023. doi: 10.1145/3571808. URL http://arxiv.org/abs/2205.13202.

Raphaël Charbey and Christophe Prieur. Stars, holes, or paths across your facebook friends: A graphlet-based characterization of many networks. *Network Science*, 2019.

Moses Charikar. Greedy Approximation Algorithms for Finding Dense Components in a Graph. 2000.

David Chavalarias and Jean-Philippe Cointet. Phylomemetic patterns in science evolution—the rise and fall of scientific fields. *PloS one*, 2013.

Norishige Chiba and Takao Nishizeki. Arboricity and Subgraph Listing Algorithms. *SIAM J. Comput.*, 1985. doi: 10.1137/0214017. URL http://epubs.siam.org/doi/10.1137/0214017.

Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On compressing social networks. In *KDD*, 2009. doi: 10.1145/1557019.1557049. URL http://portal.acm.org/citation.cfm?doid=1557019.1557049.

Sarvenaz Choobdar, Pedro Ribeiro, Sylwia Bugla, and Fernando Silva. Comparison of co-authorship networks across scientific fields using motifs. In *ASONAM*, 2012.

Shumo Chu and James Cheng. Triangle listing in massive networks and its applications. KDD, 2011. URL https://doi.org/10.1145/2020408.2020513.

John Cieslewicz and K. Ross. Database optimizations for modern hardware. 2008.

Aaron Clauset, Cosma Rohilla Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Rev.*, 2009. doi: 10.1137/070710111. URL http://arxiv.org/abs/0706.1062.

Colin B. Clement, Matthew Bierbaum, Kevin P. O'Keeffe, and Alexander A. Alemi. On the Use of ArXiv as a Dataset, 2019. URL http://arxiv.org/abs/1905.00075.

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. 2009.

Derek G. Corneil, Feodor F. Dragan, and Ekkehard Köhler. On the power of bfs to deter-

mine a graph's diameter. 2003. doi: https://doi.org/10.1002/net.10098.

Pilu Crescenzi, Roberto Grossi, Michel Habib, Leonardo Lanzi, and Andrea Marino. On computing the diameter of real-world undirected graphs. *Theoretical Computer Science*, 2013. URL https://linkinghub.elsevier.com/retrieve/pii/S0304397512008687.

Haochuan Cui, Lingfei Wu, and James A. Evans. Aging Scientists and Slowed Advance, 2022. URL http://arxiv.org/abs/2202.04044.

E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. ACM, 1969. URL https://doi.org/10.1145/800195.805928.

Maximilien Danisch, T.-H. Hubert Chan, and Mauro Sozio. Large Scale Density-friendly Graph Decomposition via Convex Programming. WWW, 2017. doi: 10.1145/3038912.3052619. URL https://dl.acm.org/doi/10.1145/3038912.3052619.

Maximilien Danisch, Oana Balalau, and Mauro Sozio. Listing k-cliques in Sparse Real-World Graphs*. In *WWW*, 2018. doi: 10.1145/3178876.3186125. URL http://dl.acm.org/citation.cfm?doid=3178876.3186125.

Maximilien Danisch, Ioannis Panagiotas, and Lionel Tabourier. Compressing bipartite graphs with a dual reordering scheme. Technical report, arXiv, 2022. URL http://arxiv.org/abs/2209.12062.

Alane M. de Lima, Murilo V. G. da Silva, and André L. Vignatti. Estimating the Clustering Coefficient Using Sample Complexity Analysis. In *LATIN*, 2022. URL https://doi.org/10.1007/978-3-031-20624-5_20.

Laxman Dhulipala, Igor Kabiljo, Brian Karrer, Giuseppe Ottaviano, Sergey Pupyrev, and Alon Shalita. Compressing Graphs and Indexes with Recursive Graph Bisection. *KDD*, 2016. doi: 10.1145/2939672.2939862. URL http://arxiv.org/abs/1602.08820.

Irit Dinur and Samuel Safra. On the Hardness of Approximating Minimum Vertex Cover. *Annals of Mathematics*, 2005. URL https://www.jstor.org/stable/3597377.

Shlomi Dolev and Marina Sadetsky. Heuristic Certificates via Approximations. In *PDCAT*, 2009. doi: 10.1109/PDCAT.2009.15.

Feodor Dragan, Michel Habib, and Laurent Viennot. Revisiting Radius, Diameter, and all Eccentricity Computation in Graphs through Certificates. Technical report, arXiv, 2018. URL http://arxiv.org/abs/1803.04660.

Mikhail Drobyshevskiy and Denis Turdakov. Random Graph Modeling: A Survey of the Concepts. *ACM Computing Surveys*, 2020. doi: 10.1145/3369782. URL https://dl.acm.org/doi/10.1145/3369782.

Ran Duan and Seth Pettie. Linear-Time Approximation for Maximum Weight Matching. *Journal of the ACM*, 2014. doi: 10.1145/2529989. URL https://dl.acm.org/doi/10.1145/2529989.

M. Ayaz Dzulfikar, Johannes K. Fichte, and Markus Hecher. The PACE 2019 Parameterized Algorithms and Computational Experiments Challenge: The Fourth Iteration. Technical report, 2019. URL http://drops.dagstuhl.de/opus/volltexte/2019/11486/.

Josep Díaz, Jordi Petit, and Maria Serna. A survey of graph layout problems. *ACM Computing Surveys*, 2002. doi: 10.1145/568522.568523. URL https://dl.acm.org/doi/10.1145/568522.568523.

Jack Edmonds. Paths, Trees, and Flowers. *Canadian Journal of Mathematics*, 1965. URL https://doi.org/10.4153/CJM-1965-045-4.

W. Ellens and R. E. Kooij. Graph measures and network robustness. Technical report, arXiv, 2013. URL http://arxiv.org/abs/1311.5064.

David Eppstein, Maarten Löffler, and Darren Strash. Listing All Maximal Cliques in Large Sparse Real-World Graphs. *ACM J. Exp. Algorithmics*, 2013. doi: 10.1145/2543629. URL https://dl.acm.org/doi/10.1145/2543629.

Erdős and Rényi. On the evolution of random graphs. 1960. URL http://snap.stanford.edu/class/cs224w-readings/erdos60random.pdf.

Yixiang Fang, Wensheng Luo, and Chenhao Ma. Densest subgraph discovery on large graphs: applications, challenges, and techniques. *VLDB*, 2022. doi: 10.14778/3554821.3554895. URL https://dl.acm.org/doi/10.14778/3554821.3554895.

Katherine Faust. A puzzle concerning triads in social networks: Graph constraints and the triad census. *Social Networks*, 2010.

Michael R. Fellows, Lars Jaffke, Aliz Izabella Király, Frances A. Rosamond, and Mathias Weller. What Is Known About Vertex Cover Kernelization? In *Adventures Between Lower Bounds and Higher Altitudes*. 2018. URL http://link.springer.com/10.1007/978-3-319-98355-4_19.

M. R. Ferreira, N. Reisz, W. Schueller, V. D. P. Servedio, S. Thurner, and V. Loreto. *Quantifying Exaptation in Scientific Evolution*. 2020. doi: 10.1007/978-3-030-45784-6_5.

Santo Fortunato, Carl T Bergstrom, Katy Börner, James A Evans, Dirk Helbing, Staša Milojević, Alexander M Petersen, Filippo Radicchi, Roberta Sinatra, Brian Uzzi, et al. Science of science. *Science*, 2018.

Bailey K. Fosdick, Daniel B. Larremore, Joel Nishimura, and Johan Ugander. Configuring Random Graph Models with Fixed Degree Sequences. Technical report, arXiv, 2017. URL http://arxiv.org/abs/1608.00607.

Linton Freeman. A Set of Measures of Centrality Based on Betweenness. *Sociometry*, 1977. doi: 10.2307/3033543.

M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1976. doi: 10.1016/0304-3975(76)90059-1. URL https://www.sciencedirect.com/science/article/pii/0304397576900591.

Mikael Gast and Mathias Hauptmann. Approximability of the vertex cover problem in power-law graphs. *Theoretical Computer Science*, 2014. doi: 10.1016/j.tcs.2013.11.004. URL https://www.sciencedirect.com/science/article/pii/S0304397513008220.

Mohsen Ghaffari, Themis Gouleakis, Christian Konrad, Slobodan Mitrović, and Ronitt Rubinfeld. Improved Massively Parallel Computation Algorithms for MIS, Matching, and Vertex Cover. PODC, 2018. URL https://doi.org/10.1145/3212734.3212743.

A. V. Goldberg. Finding a Maximum Density Subgraph. Technical Report, University of California at Berkeley, 1984.

Fernando C. Gomes, Cláudio N. Meneses, Panos M. Pardalos, and Gerardo Valdisio R. Viana. Experimental Analysis of Approximation Algorithms for the Vertex Cover and Set Covering Problems. *Computers & Operations Research*, 2006. doi: 10.1016/j.cor.2005.03.030. URL https://www.sciencedirect.com/science/article/pii/S0305054805001255.

Michael T. Goodrich and Pawel Pszona. External-Memory Network Analysis Algorithms for Naturally Sparse Graphs. Technical report, arXiv, 2011. URL http://arxiv.org/abs/1106.6336.

Xiangyang Gou and Lei Zou. Sliding Window-based Approximate Triangle Counting over Streaming Graphs with Duplicate Edges. In *PODS/SIGMOD*. 2021. URL https://doi.org/10.1145/3448016.3452800.

M. M. Halldórsson and J. Radhakrishnan. Greed is good: Approximating independent

sets in sparse and bounded-degree graphs. *Algorithmica*, 1997. URL https://doi.org/10.1007/BF02523693.

L. H. Harper. Optimal Assignments of Numbers to Vertices. *Journal of the Society for Industrial and Applied Mathematics*, 1964. doi: 10.1137/0112012. URL http://epubs.siam.org/doi/10.1137/0112012.

Demian Hespe, Sebastian Lamm, Christian Schulz, and Darren Strash. WeGotYouCovered: The Winning Solver from the PACE 2019 Challenge, Vertex Cover Track. In *CSC*. 2020. URL https://epubs.siam.org/doi/10.1137/1.9781611976229.1. Code available at https://zenodo.org/record/2816116.

Thomas T Hills. Animal foraging and the evolution of goal-directed cognition. *Cognitive science*, 2006.

Thomas T Hills, Michael N Jones, and Peter M Todd. Optimal foraging in semantic memory. *Psychological review*, 2012.

Thomas T Hills, Peter M Todd, David Lazer, A David Redish, Iain D Couzin, Cognitive Search Research Group, et al. Exploration versus exploitation in space, mind, and society. *Trends in cognitive sciences*, 2015.

Dorit S. Hochbaum. The t-vertex cover problem: Extending the half integrality framework with budget constraints. In *Approximation Algorithms for Combinatiorial Optimization*, 1998. doi: 10.1007/BFb0053968.

Constance Holden. Federation of Scientists Plans "Great Leap Forward". *Science*, 1974. doi: 10.1126/science.185.4145.47.a. URL https://www.science.org/doi/10.1126/science.185.4145.47.a.

Paul W Holland and Samuel Leinhardt. Local structure in social networks. *Sociological methodology*, 1976.

Petter Holme, Beom Jun Kim, Chang No Yoon, and Seung Kee Han. Attack vulnerability of complex networks. *Physical Review E*, 2002. doi: 10.1103/PhysRevE.65.056109. URL http://arxiv.org/abs/cond-mat/0202410.

Lin Hu, Lei Zou, and Yu Liu. Accelerating Triangle Counting on GPU. SIGMOD/PODS, 2021. URL https://doi.org/10.1145/3448016.3452815.

Tianran Hu, Jiebo Luo, and Wei Liu. Life in the "matrix": Human mobility patterns in the cyber space. In *Conference on Web and Social Media*, 2018.

The Dang Huynh, Fabien Mathieu, and Laurent Viennot. LiveRank: How to Refresh Old Crawls. In *Algorithms and Models for the Web Graph*. 2014. doi: 10.1007/978-3-319-13123-8_12. URL http://link.springer.com/10.1007/978-3-319-13123-8_12.

Iacopo Iacopini, Staša Milojević, and Vito Latora. Network dynamics of innovation processes. *Phys. Rev. Lett.*, 2018. doi: 10.1103/PhysRevLett.120.048301. URL http://arxiv.org/abs/1707.04239.

Alon Itai and Michael Rodeh. Finding a minimum circuit in a graph. STOC, 1977. URL https://doi.org/10.1145/800105.803390.

Swami Iyer, Timothy Killingback, Bala Sundaram, and Zhen Wang. Attack Robustness and Centrality of Complex Networks. *PLOS ONE*, 2013. URL https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0059613.

Tao Jia, Dashun Wang, and Boleslaw K Szymanski. Quantifying patterns of research-interest evolution. *Nature Human Behaviour*, 2017.

Jan Kara, Jan Kratochvıl, and David R Wood. On the complexity of the balanced vertex ordering problem. 2007.

George Karakostas. A Better Approximation Ratio for the Vertex Cover Problem. In *Automata, Languages and Programming*, 2005. doi: 10.1007/11523468_84.

Konstantinos I. Karantasis, Andrew Lenharth, Donald Nguyen, Mara J. Garzaran, and Keshav Pingali. Parallelization of Reordering Algorithms for Bandwidth and Wavefront Reduction. In *SC: High Performance Computing, Networking, Storage and Analysis*, 2014. doi: 10.1109/SC.2014.80. URL http://ieeexplore.ieee.org/document/7013062/.

Richard M. Karp. Reducibility among Combinatorial Problems. In *Complexity of Computer Computations*. 1972. URL https://doi.org/10.1007/978-1-4684-2001-2_9.

Brian Karrer and M. E. J. Newman. Random graphs containing arbitrary distributions of subgraphs. *Physical Review E*, 2010. doi: 10.1103/PhysRevE.82.066118. URL http://arxiv.org/abs/1005.1659.

George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 1998. doi: 10.1137/S1064827595287997. URL http://epubs.siam.org/doi/10.1137/S1064827595287997.

Subhash Khot. On the power of unique 2-prover 1-round games. STOC, 2002. URL https://doi.org/10.1145/509907.510017.

Subhash Khot and Oded Regev. Vertex cover might be hard to approximate to within 2-epsilon. *Journal of Computer and System Sciences*, 2008. doi: 10.1016/j.jcss.2007.06.019. URL https://linkinghub.elsevier.com/retrieve/pii/S0022000007000864.

J. Klafter and I. M. Sokolov. First steps in random walks: From tools to applications. 2011. doi: 10.1093/acprof:oso/9780199234868.001.0001.

Jon Kleinberg and Eva Tardos. *Algorithm Design*. 2006. URL https://archive.org/details/AlgorithmDesign1stEditionByJonKleinbergAndEvaTardos2005PDF/page/n429.

Thomas S. Kuhn. *The Essential Tension: Selected Studies in Scientific Tradition and Change*. 1979. URL https://press.uchicago.edu/ucp/books/book/chicago/E/bo5970650.html.

Jérôme Kunegis. Konect: The koblenz network collection. WWW Companion, 2013. doi: 10.1145/2487788.2488173. URL http://konect.cc.

Kazuhiro Kurita, Kunihiro Wasa, Hiroki Arimura, and Takeaki Uno. Efficient Enumeration of Dominating Sets for Sparse Graphs. *arXiv*, 2018. doi: 10.4230/LIPIcs.ISAAC.2018.8. URL http://arxiv.org/abs/1802.07863.

Yann Lanoiselée and Denis S. Grebenkov. Unraveling intermittent features in single-particle trajectories by a local convex hull method. *Physical Review E*, 2017. doi: 10.1103/PhysRevE.96.022144. URL https://link.aps.org/doi/10.1103/PhysRevE.96.022144.

Matthieu Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science*, 2008. doi: 10.1016/j.tcs.2008.07.017. URL https://linkinghub.elsevier.com/retrieve/pii/S0304397508005392.

Fabrice Lécuyer, Maximilien Danisch, and Lionel Tabourier. [Re] Speedup graph processing by graph ordering. *The ReScience journal*, 2021. URL https://doi.org/10.5281/zenodo.4836230.

Fabrice Lécuyer, Louis Jachiet, Clémence Magnien, and Lionel Tabourier. Tailored vertex ordering for faster triangle listing in large graphs. In *ALENEX*, 2023a. URL https://arxiv.org/abs/2203.04774.

Fabrice Lécuyer, Lionel Tabourier, and Clémence Magnien. Quality certification of vertex cover heuristics on real-world networks. 2023b.

Victor E. Lee, Ning Ruan, Ruoming Jin, and Charu Aggarwal. A Survey of Algorithms for Dense Subgraph Discovery. In *Managing and Mining Graph Data*. 2010. doi: 10.1007/

978-1-4419-6045-0_10. URL https://link.springer.com/10.1007/978-1-4419-6045-0_10.

Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection, 2014. URL http://snap.stanford.edu/data.

Jure Leskovec, Lars Backstrom, Ravi Kumar, and Andrew Tomkins. Microscopic evolution of social networks. In *KDD*, 2008.

Rong-Hua Li, Sen Gao, Lu Qin, Guoren Wang, Weihua Yang, and Jeffrey Xu Yu. Ordering heuristics for k-clique listing. *VLDB*, 2020. doi: 10.14778/3407790.3407843. URL https://dl.acm.org/doi/10.14778/3407790.3407843.

Yongsub Lim, U Kang, and Christos Faloutsos. SlashBurn: Graph Compression and Mining beyond Caveman Communities. *IEEE Trans. Knowl. Data Eng.*, 2014. doi: 10.1109/TKDE.2014.2320716. URL http://ieeexplore.ieee.org/document/6807798/.

Yiling Lin, James A Evans, and Lingfei Wu. New directions in science emerge from disconnection and discord. *Journal of Informetrics*, 2022.

Lu Liu, Yang Wang, Roberta Sinatra, C Lee Giles, Chaoming Song, and Dashun Wang. Hot streaks in artistic, cultural, and scientific careers. *Nature*, 2018.

Clemence Magnien, Matthieu Latapy, and Jean-Loup Guillaume. Impact of Random Failures and Attacks on Poisson and Power-Law Random Networks. *ACM Computing Surveys*, 2011. doi: 10.1145/1922649.1922650. URL http://arxiv.org/abs/0908.3154.

Clémence Magnien, Matthieu Latapy, and Michel Habib. Fast computation of empirically tight bounds for the diameter of massive graphs. *Journal of Experimental Algorithmics*, 2009. doi: 10.1145/1412228.1455266. URL https://dl.acm.org/doi/10.1145/1412228.1455266.

Yury Makarychev. https://cstheory.stackexchange.com/q/47020, 2020.

Kazuhisa Makino and Takeaki Uno. New Algorithms for Enumerating All Maximal Cliques. In *Algorithm Theory - SWAT 2004*. 2004. doi: 10.1007/978-3-540-27810-8_23. URL http://link.springer.com/10.1007/978-3-540-27810-8_23.

Fragkiskos Malliaros, Christos Giatsidis, Apostolos Papadopoulos, and Michalis Vazirgiannis. The Core Decomposition of Networks: Theory, Algorithms and Applications. *VLDB*, 2019. URL https://hal-centralesupelec.archives-ouvertes.fr/hal-01986309.

James G. March. Exploration and Exploitation in Organizational Learning. *Organization Science*, 1991. URL https://www.jstor.org/stable/2634940.

Daniel Margo and Margo Seltzer. A scalable distributed graph partitioner. *VLDB*, 2015. URL https://dl.acm.org/doi/10.14778/2824032.2824046.

David W. Matula and Leland L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *Journal of the ACM*, 1983. doi: 10.1145/2402.322385. URL https://dl.acm.org/doi/10.1145/2402.322385.

Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv:1802.03426*, 2018.

Silvio Micali and Vijay V. Vazirani. An O(v|v| c |E|) algoithm for finding maximum matching in general graphs. In *SFCS*, 1980. doi: 10.1109/SFCS.1980.12.

R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: simple building blocks of complex networks. *Science*, 2002. doi: 10.1126/science.298.5594.824.

Ron Milo, Shalev Itzkovitz, Nadav Kashtan, Reuven Levitt, Shai Shen-Orr, Inbal Ayzenshtat, Michal Sheffer, and Uri Alon. Superfamilies of Evolved and Designed Networks. *Sci-*

ence, 2004. doi: 10.1126/science.1089167. URL https://science.sciencemag.org/content/303/5663/1538.

Staša Milojević. Quantifying the cognitive extent of science. *Journal of Informetrics*, 2015.

Staša Milojević, Cassidy R. Sugimoto, Erjia Yan, and Ying Ding. The cognitive structure of Library and Information Science: Analysis of article title words. *Journal of the American Society for Information Science and Technology*, 2011. doi: 10.1002/asi.21602.

M. E. J. Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 2006. doi: 10.1073/pnas.0601602103. URL http://www.pnas.org/cgi/doi/10.1073/pnas.0601602103.

M. E. J. Newman. Random graphs with clustering. *Phys. Rev. Lett.*, 2009. doi: 10.1103/PhysRevLett.103.058701. URL http://arxiv.org/abs/0903.4009.

Thomas Messi Nguélé, Maurice Tchuente, and Jean-François Méhaut. Using Complex-Network properties For Efficient Graph Analysis. 2017.

Jérémy Omer and Antonio Mucherino. Referenced Vertex Ordering Problem: Theory, Applications and Solution Methods. *Open Journal of Mathematical Optimization*, 2021. doi: 10.5802/ojmo.8. URL https://hal.science/hal-02509522.

Mark Ortmann and Ulrik Brandes. Triangle Listing Algorithms: Back from the Diversion. In *ALENEX*. 2013. doi: 10.1137/1.9781611973198.1. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611973198.1.

Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web, 1999.

Raj K Pan, Alexander M Petersen, Fabio Pammolli, and Santo Fortunato. The memory of science: Inflation, myopia, and the knowledge network. *Journal of Informetrics*, 2018.

Pooja Pandey and Abraham P. Punnen. The generalized vertex cover problem and some variations. *Discrete Optimization*, 2018. doi: 10.1016/j.disopt.2018.06.004. URL https://www.sciencedirect.com/science/article/pii/S1572528617301780.

Luca Pappalardo, Filippo Simini, Salvatore Rinzivillo, Dino Pedreschi, Fosca Giannotti, and Albert-László Barabási. Returners and explorers dichotomy in human mobility. *Nature communications*, 2015.

Michael Park, Erin Leahey, and Russell J. Funk. Papers and patents are becoming less disruptive over time. *Nature*, 2023. doi: 10.1038/s41586-022-05543-x. URL https://www.nature.com/articles/s41586-022-05543-x.

Noujan Pashanasangi and Comandur Seshadhri. Faster and Generalized Temporal Triangle Counting, via Degeneracy Ordering. *arXiv*, 2021. URL http://arxiv.org/abs/2106.02762.

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine Learning in Python. *The Journal of Machine Learning Research*, 2011.

Hao Peng, Qing Ke, Ceren Budak, Daniel M Romero, and Yong-Yeol Ahn. Neural embeddings of scholarly periodicals reveal complex disciplinary organizations. *Science Advances*, 2021.

Ali Pinar, Comandur Seshadhri, and Vaidyanathan Vishal. ESCAPE: Efficiently Counting All 5-Vertex Subgraphs. In *WWW*, 2017. doi: 10.1145/3038912.3052597. URL https://dl.acm.org/doi/10.1145/3038912.3052597.

Nataša Pržulj. Biological network comparison using graphlet degree distribution. *Bioinformatics*, 2007. URL https://doi.org/10.1093/bioinformatics/btl301.

Usha Nandini Raghavan, Reka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Phys. Rev. E*, 2007. doi: 10.1103/PhysRevE.76.036106. URL http://arxiv.org/abs/0709.2938.

Fabiano L. Ribeiro and Diego Rybski. Mathematical models to explain the origin of urban scaling laws: a synthetic review, 2021. URL http://arxiv.org/abs/2111.08365.

Ryan A. Rossi and Nesreen K. Ahmed. Coloring Large Complex Networks. Technical report, 2014. URL http://arxiv.org/abs/1403.3448.

Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015. URL https://networkrepository.com.

Mikhail Rudoy. https://cstheory.stackexchange.com/q/38274, 2017.

Ilya Safro and Boris Temkin. Multiscale approach for the network compression-friendly ordering. *Journal of Discrete Algorithms*, 2011. doi: 10.1016/j.jda.2010.09.007. URL https://www.sciencedirect.com/science/article/pii/S1570866710000407.

Pedro V. Sander, Diego Nehab, Eden Chlamtac, and Hugues Hoppe. Efficient traversal of mesh edges using adjacency primitives. *ACM Trans. Graph.*, 2008. doi: 10.1145/1409060.1409097. URL https://dl.acm.org/doi/10.1145/1409060.1409097.

Thomas J. Schaefer. The complexity of satisfiability problems. STOC, 1978. URL https://doi.org/10.1145/800133.804350.

Thomas Schank and Dorothea Wagner. Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study. In *Experimental and Efficient Algorithms*. 2005. doi: 10.1007/11427186_54. URL http://link.springer.com/10.1007/11427186_54.

Markus Schläpfer, Lei Dong, Kevin O'Keeffe, Paolo Santi, Michael Szell, Hadrien Salat, Samuel Anklesaria, Mohammad Vazifeh, Carlo Ratti, and Geoffrey B West. The universal visitation law of human mobility. *Nature*, 2021.

Christian M Schneider, Vitaly Belik, Thomas Couronné, Zbigniew Smoreda, and Marta C González. Unravelling daily human mobility motifs. *Journal of The Royal Society Interface*, 2013.

Vedran Sekara, Pierre Deville, Sebastian E Ahnert, Albert-László Barabási, Roberta Sinatra, and Sune Lehmann. The chaperone effect in scientific publishing. *Proceedings of the National Academy of Sciences*, 2018.

Martyn L Senior. From gravity modelling to entropy maximizing: a pedagogic guide. *Progress in Human Geography*, 1979.

Comandur Seshadhri, Ali Pinar, and Tamara G. Kolda. Wedge Sampling for Computing Clustering Coefficients and Triangle Counts on Large Graphs. *Statistical Analy Data Mining*, 2014. doi: 10.1002/sam.11224. URL http://arxiv.org/abs/1309.3321.

Jessica Shi, Laxman Dhulipala, and Julian Shun. Parallel Clique Counting and Peeling Algorithms. *arXiv*, 2021. URL http://arxiv.org/abs/2002.10047.

Uri Shwed and Peter S. Bearman. The Temporal Structure of Scientific Consensus Formation. *Am Sociol Rev*, 2010. URL https://doi.org/10.1177/0003122410388488.

Filippo Simini, Marta C González, Amos Maritan, and Albert-László Barabási. A universal model for mobility and migration patterns. *Nature*, 2012.

Filippo Simini, Gianni Barlacchi, Massimilano Luca, and Luca Pappalardo. A deep gravity model for mobility flows generation. *Nature communications*, 2021.

Georg Simmel. *Soziologie*. 1908.

Roberta Sinatra, Dashun Wang, Pierre Deville, Chaoming Song, and Albert-László Barabási. Quantifying the evolution of individual scientific impact. *Science*, 2016.

Chakresh Kumar Singh, Emma Barme, Robert Ward, Liubov Tupikina, and Marc Santolini. Quantifying the rise and fall of scientific fields. *PloS one*, 2022.

Chakresh Kumar Singh, Liubov Tupikina, Fabrice Lécuyer, Michele Starnini, and Marc Santolini. Charting mobility patterns in the scientific knowledge landscape. *arXiv preprint arXiv*, 2023. URL https://arxiv.org/abs/2302.13054.

Stavros Sintos and Panayiotis Tsaparas. Using strong triadic closure to characterize ties in social networks. In *KDD*, 2014.

Tom A.B. Snijders. Statistical Models for Social Networks. *Annual Review of Sociology*, 2011. URL https://doi.org/10.1146/annurev.soc.012809.102709.

Olaf Sporns and Rolf Kötter. Motifs in brain networks. *PLOS Biology*, 2004. doi: 10.1371/journal.pbio.0020369. URL https://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.0020369.

Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *KDD*, 2012. doi: 10.1145/2339530.2339722. URL http://dl.acm.org/citation.cfm?doid=2339530.2339722.

Isabelle Stanton and Ali Pinar. Constructing and Sampling Graphs with a Prescribed Joint Degree Distribution. Technical report, arXiv, 2011. URL http://arxiv.org/abs/1103.4875.

Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. VEBO: A Vertex- and Edge-Balanced Ordering Heuristic to Load Balance Parallel Graph Processing. *arXiv*, 2018. URL http://arxiv.org/abs/1806.06576.

Ye Sun and Vito Latora. The evolution of knowledge within and across fields in modern physics. *Scientific reports*, 2020.

E. Sáenz-de Cabezón and H.P. Wynn. Measuring the robustness of a network using minimal vertex covers. *Mathematics and Computers in Simulation*, 2014. doi: 10.1016/j.matcom.2014.04.001. URL https://linkinghub.elsevier.com/retrieve/pii/S0378475414000731.

Lionel Tabourier, Camille Roth, and Jean-Philippe Cointet. Generating constrained random graphs using multiple edge switches. *Journal of Experimental Algorithmics*, 2011. doi: 10.1145/1963190.2063515. URL http://lioneltabourier.fr/documents/Tabourier_Generating_Random_Graphs.pdf.

Frank W. Takes and Walter A. Kosters. Determining the diameter of small world networks. CIKM, 2011. URL https://doi.org/10.1145/2063576.2063748.

Kanat Tangwongsan, A. Pavan, and Srikanta Tirthapura. Parallel Triangle Counting in Massive Streaming Graphs. Technical report, arXiv, 2013. URL http://arxiv.org/abs/1308.2166.

Robert Tarjan. Depth-first search and linear graph algorithms. 1972. doi: 10.1137/0201010.

Nikolaj Tatti and Aristides Gionis. Density-friendly Graph Decomposition. WWW, 2015. URL https://doi.org/10.1145/2736277.2741119.

Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T. Vo. The more the merrier: efficient multi-source graph traversal. *VLDB*, 2014. doi: 10.14778/2735496.2735507. URL https://dl.acm.org/doi/10.14778/2735496.2735507.

Charalampos Tsourakakis. The K-clique Densest Subgraph Problem. In *WWW*, 2015. doi: 10.1145/2736277.2741098. URL https://dl.acm.org/doi/10.1145/2736277.2741098.

Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. FENNEL: streaming graph partitioning for massive scale graphs. In *WSDM*, 2014. doi: 10.1145/2556195.2556213. URL https://dl.acm.org/doi/10.1145/2556195.2556213.

Tuninetti, Marta, Aleta, Alberto, Paolotti, Daniela, Moreno, Yamir, and Starnini, Michele. Prediction of new scientific collaborations through multiplex networks. *EPJ Data Sci.*, 2021. URL https://doi.org/10.1140/epjds/s13688-021-00282-x.

Ata Turk and Duru Turkoglu. Revisiting Wedge Sampling for Triangle Counting. WWW, 2019. URL https://doi.org/10.1145/3308558.3313534.

Krzysztof Turowski, Jithin K. Sreedharan, and Wojciech Szpankowski. Temporal Ordered Clustering in Dynamic Networks: Unsupervised and Semi-supervised Learning Algorithms. 2020. URL http://arxiv.org/abs/1905.00672.

TutorialsPoint. Quick guide to cuda. URL http://www.tutorialspoint.com/cuda/cuda_quick_guide.htm.

Enrico Ubaldi, Bernardo Monechi, Claudio Chiappetta, and Vittorio Loreto. Heterogeneity and segregation of mobility patterns. *Handbook on Entropy, Complexity and Spatial Dynamics*, 2021. URL https://www.elgaronline.com/view/edcoll/9781839100581/9781839100581.00038.xml.

Takeaki Uno. Implementation issues of clique enumeration algorithm. *Prog. Inform.*, 2012. doi: 10.2201/NiiPi.2012.9.5. URL http://www.nii.ac.jp/pi/n9/9_25.html.

Sergi Valverde and Ricard V. Solé. Network motifs in computational graphs: A case study in software architecture. *Phys. Rev. E*, 2005. doi: 10.1103/PhysRevE.72.026107. URL https://link.aps.org/doi/10.1103/PhysRevE.72.026107.

Laurens Van der Maaten. Learning a parametric embedding by preserving local structure. In *Artificial intelligence and statistics*. PMLR, 2009.

Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 2008.

Katherine Van Koevering, Austin Benson, and Jon Kleinberg. Random Graphs with Prescribed K-Core Sequences: A New Null Model for Network Analysis. In *WWW*, 2021. doi: 10.1145/3442381.3450001. URL https://dl.acm.org/doi/10.1145/3442381.3450001.

Sebastiano Vigna and Paolo Boldi. Axioms for Centrality. *Internet Mathematics*, 2014. doi: 10.1080/15427951.2013.865686. URL https://www.internetmathematicsjournal.com/article/1564.

Dashun Wang, Chaoming Song, and Albert-László Barabási. Quantifying long-term scientific impact. *Science*, 2013.

Leyuan Wang, Yangzihao Wang, Carl Yang, and John D. Owens. A Comparative Study on Exact Triangle Counting Algorithms on the GPU. In *Workshop on High Performance Graph Processing*, 2016. doi: 10.1145/2915516.2915521. URL https://dl.acm.org/doi/10.1145/2915516.2915521.

Yingfan Wang, Haiyang Huang, Cynthia Rudin, and Yaron Shaposhnik. Understanding how dimension reduction tools work: An empirical approach to deciphering t-sne, umap, trimap, and pacmap for data visualization. *J. Mach. Learn. Res.*, 2021.

Stanley Wasserman, Katherine Faust, et al. Social network analysis: Methods and applications. 1994.

Duncan Watts and Steven Strogatz. Collective Dynamics of Small-World Networks. 1998.

Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. Speedup Graph Processing by Graph Ordering. SIGMOD/PODS'16, 2016. doi: 10.1145/2882903.2915220. URL https://dl.acm.org/doi/10.1145/2882903.2915220.

Barry Wellman. Structural analysis: From method and metaphor to theory and substance. *Contemporary Studies in Sociology*, 1988.

A. G. Wilson. A statistical theory of spatial distribution models. *Transportation Research*, 1967. doi: 10.1016/0041-1647(67)90035-4. URL https://www.sciencedirect.com/science/article/pii/0041164767900354.

Alan Wilson. *Entropy in Urban and Regional Modelling* (*Routledge Revivals*). 2013.

Laiyun Wu, Samiul Hasan, Younshik Chung, and Jee Eun Kang. Understanding the Heterogeneity of Human Mobility Patterns: User Characteristics and Modal Preferences. *Sustainability*, 2021. doi: 10.3390/su132413921. URL https://www.mdpi.com/2071-1050/13/24/13921.

Lingfei Wu, Dashun Wang, and James A. Evans. Large teams develop and small teams disrupt science and technology. *Nature*, 2019. doi: 10.1038/s41586-019-0941-9. URL https://www.nature.com/articles/s41586-019-0941-9.

Stefan Wuchty, Benjamin F Jones, and Brian Uzzi. The increasing dominance of teams in production of knowledge. *Science*, 2007.

Di Xiao, Yi Cui, Daren B.H. Cline, and Dmitri Loguinov. On Asymptotic Cost of Triangle Listing in Random Graphs. In *SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 2017. doi: 10.1145/3034786.3034790. URL https://dl.acm.org/doi/10.1145/3034786.3034790.

Mingyu Xiao, Sen Huang, Yi Zhou, and Bolin Ding. Efficient Reductions and a Fast Algorithm of Maximum Weighted Independent Set. WWW, 2021. URL https://doi.org/10.1145/3442381.3450130.

Yasin Yigit, Vahid Khalilpour Akram, and Orhan Dagdeviren. Breadth-first search tree integrated vertex cover algorithms for link monitoring and routing in wireless sensor networks. *Computer Networks*, 2021. doi: 10.1016/j.comnet.2021.108144. URL https://linkinghub.elsevier.com/retrieve/pii/S1389128621002103.

Yian Yin and Dashun Wang. The time dimension of science: Connecting the past to the future. *Journal of Informetrics*, 2017. doi: 10.1016/j.joi.2017.04.002. URL https://www.sciencedirect.com/science/article/pii/S1751157717300020.

Qiu Fang Ying, Srinivasan Venkatramanan, and Dah Ming Chiu. Modeling and Analysis of Scholar Mobility on Scientific Landscape, 2015. URL http://arxiv.org/abs/1502.00523.

Neal E. Young and Louis Jachiet. https://cstheory.stackexchange.com/q/48572, 2020.

An Zeng, Zhesi Shen, Jianlin Zhou, Ying Fan, Zengru Di, Yougui Wang, H. Eugene Stanley, and Shlomo Havlin. Increasing trend of scientists to switch between topics. *Nature Communications*, 2019. doi: 10.1038/s41467-019-11401-8. URL https://www.nature.com/articles/s41467-019-11401-8.

Zhi-Dan Zhao, Zi-Gang Huang, Liang Huang, Huan Liu, and Ying-Cheng Lai. Scaling and correlation of human movements in cyberspace and physical space. *Physical Review E*, 2014.

Xiao Zhou and Takao Nishizeki. Edge-coloring and f-coloring for various classes of graphs. In *Algorithms and Computation*, 1994. doi: 10.1007/3-540-58325-4_182.