Automatic classification of dynamic graphs

Mohammed Yessin Neggaz

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET INFORMATIQUE

PAR

MOHAMMED YESSIN NEGGAZ

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

AUTOMATIC CLASSIFICATION OF DYNAMIC GRAPHS

Thèse encadrée par :

Arnaud CASTEIGTS (co-encadrant), Serge CHAUMETTE (co-directeur)
et Colette JOHNEN (co-directrice)

Soutenue le 24 octobre 2016

Devant la commission d'examen composée de :

CASTEIGTS, Arnaud	MC., Univ. de Bordeaux	Co-encadrant
CHAUMETTE, Serge	PR., Univ. de Bordeaux	Co-directeur
GUINAND, Frédéric	PR., Univ. Le Havre	Rapporteur et Président
JOHNEN, Colette	PR., Univ. de Bordeaux	Co-directrice
SOHIER, Devan	MC., Univ. de Versailles UVSQ	Examinateur
VILLAIN, Vincent	PR., Univ. de Picardie Jules Verne	Rapporteur

– 2016 –

Acknowledgments

It is a great pleasure to thank the many people who supported me during my Ph.D. studies.

I would first like to express my sincere thanks to Arnaud Casteigts, Serge Chaumette and Colette Johnen for their support throughout these three years of thesis and well before that. It is thanks to their pedagogy and their open-mindedness that I chose this way during my master's degree, and it is thanks to their help and guidance that I was able to go all the way. I cannot say how valuable our exchanges and their advice have been. I enjoyed a lot their positive attitude to make a friendly work atmosphere.

I thank a lot Ralf Klasing and Joseph Peters for having involved me in several works. I really appreciated the opportunity to collaborate with them. Our discussions, always very fruitful, have greatly influenced the direction of my research and the completion of these three years.

I warmly thank Frédéric Guinand and Vincent Villain for agreeing to be the reviewers of my thesis and for their careful reading. Their comments and suggestions, all very constructive, helped me a lot. I am very grateful to Devan Sohier for participating in my thesis jury and thank him sincerely.

I would also like to thank my colleagues, both Vincent, Matthieu, Christelle, David, Antoine, Sebastien, Daouda, Martin, Duy and Jigar for their sympathy and good humor during these three years. Thanks to all the members of the LaBRI and ENSEIRB-MATMECA with whom I have had productive exchanges.

I thank all the teachers that I had during my academic career and who encouraged me to go further. A special thought for Malika, she was the best.

Many thanks to my friends who always supported me and with whom I had a pleasant time. Thanks to Omar (Reda Hlou), Nadia, without forgetting their little Assyia, Nesrine, Wejdi, Akram, (little) Omar, Fethi (3ayrad), Soukaina, Massi, Khelaf, Carvajal (not to forget anyone), Salma, Charlotte, Mohamed, Samia, Rachida, Kawther, Médine (better known as Munevver), Sadetdin, Fatih and their parents. They greatly alleviated the stress experienced while writing this thesis. A special thanks to Omar (Moulkhaloua), Riazi, Benallel, Mhimda, Zino, Ghoul, Vansalah and Benkrama family.

Of course, I can not conclude without thanking my family wholeheartedly, my parents and my grandmother, my brothers and sisters who, during the three years of thesis, have always supported and encouraged me, as usual.

I learned during these three years that the only way in order not to miss any case and to visit all the possibilities is to present things formally. I therefore propose a formal description of these acknowledgments.

Formal presentation of acknowledgments

The set of people and their interactions can be represented as a dynamic graph $\mathcal{G} = (V, \{E_i\})$ where $V = \cup\{friends, advisor, teachers, colleagues, family, reviewers, jury\}$. A solution to the problem would be to use a predicate `thanked` directly for all the nodes of the graph. This centralized algorithm, which can be described simply as follows: $\forall$ `person` $\in V$, `thanked(me,person)` $= true$, has as a condition for the success of its execution the fact that everyone must read this document. This limits its use to categories of graphs that do not correspond to real practical contexts, while a distributed approach could make the problem solvable in a more realistic models. A dynamic graph representing a social network of this type is often part of the class *source connectivity* (Def. 1.13) where there exist at least one node from which there is a journey to all other nodes (*Property 1*). I propose bellow a distributed event-based acknowledgments algorithm that uses the propagation of information in dynamic graphs. *Property 1* is proved to be a necessary and sufficient condition for the success of this algorithm (in this class of graphs, everyone will be thanked if at least one person reads this document). See Section 1.2.1 for details.

Distributed acknowledgments algorithm executed on every node $v \in V$:

```
1  onReading this document:
2  |  source ← author;
3  |  thank(source, v);
4  |  thanked ← true;

5  onContact with person ∈ V\{v}:
6  |  if thanked ∧ person ∉ thankedNodes then
7  |  |  send(source);
8  |  |  add(person,thankedNodes);

9  onReception of message m:
10 |  if ¬ thanked then
11 |  |  source ← getContent(m);
12 |  |  thank(source, v);
13 |  |  thanked ← true;
```

Title Automatic Classification of Dynamic Graphs.

Abstract Dynamic networks consist of entities making contact over time with one another. A major challenge in dynamic networks is to predict mobility patterns and decide whether the evolution of the topology satisfies requirements for the success of a given algorithm. The types of dynamics resulting from these networks are varied in scale and nature. For instance, some of these networks remain connected at all times; others are always disconnected but still offer some kind of connectivity over time and space (*temporal connectivity*); others are recurrently connected, periodic, *etc.* All of these contexts can be represented as dynamic graph classes corresponding to necessary or sufficient conditions for given distributed problems or algorithms. Given a dynamic graph, a natural question to ask is to which of the classes this graph belongs. In this work we provide a contribution to the automation of dynamic graphs classification. We provide strategies for testing membership of a dynamic graph to a given class and a generic framework to test properties in dynamic graphs. We also attempt to understand what can still be done in a context where no property on the graph is guaranteed through the distributed problem of maintaining a spanning forest in highly dynamic graphs.

Titre Classification automatique de graphes dynamiques.

Résumé Les réseaux dynamiques sont constitués d'entités établissant des contacts les unes avec les autres dans le temps. Un défi majeur dans les réseaux dynamiques est de prédire les modèles de mobilité et de décider si l'évolution de la topologie satisfait aux exigences du succès d'un algorithme donné. Les types de dynamique résultant de ces réseaux sont variés en échelle et en nature. Par exemple, certains de ces réseaux restent connexes tout le temps; d'autres sont toujours déconnectés mais offrent toujours une sorte de connexité dans le temps et dans l'espace (connexité temporelle); d'autres sont connexes de manière récurrente, périodique, *etc*. Tous ces contextes peuvent être représentés sous forme de classes de graphes dynamiques correspondant à des conditions nécessaires et/ou suffisantes pour des problèmes ou algorithmes distribués donnés. Étant donné un graphe dynamique, une question naturelle est de savoir à quelles classes appartient ce graphe. Dans ce travail, nous apportons une contribution à l'automatisation de la classification de graphes dynamiques. Nous proposons des stratégies pour tester l'appartenance d'un graphe dynamique à une classe donnée et nous définissons un cadre générique pour le test de propriétés dans les graphes dynamiques. Nous explorons également le cas où aucune propriété sur le graphe n'est garantie, à travers l'étude du problème de maintien d'une forêt d'arbres couvrants dans un graphe dynamique.

Keywords Dynamic graphs, dynamic networks, delay-tolerant networks, mobile networks, evolving graphs, time-varying graphs, graphs, journeys, classes, classification, connectivity, graph algorithms, distributed algorithms.

Mots-clés Graphes dynamiques, réseaux dynamiques, réseaux mobiles, systèmes répartis dynamiques, graphes évolutifs, graphes variants dans le temps, graphes, trajets, classes, classification, connexité, algorithmes sur les graphes, algorithmes distribués.

Laboratoire d'accueil Laboratoire Bordelais de Recherche en Informatique.
351, cours de la Libération, 33405 Talence.

Résumé long

Parmi les évolutions majeures dans le domaine de l'informatique, on note l'émergence des réseaux dynamiques. Ces réseaux sont constitués d'entités entrant en contact les unes avec les autres dans le temps, ce qui les différencie des réseaux statiques où la topologie reste inchangée. Plus tôt, plusieurs études ont été faites dans cette discipline pour développer de nouvelles techniques, modèles et analyses afin d'étudier et de résoudre des problèmes importants dans un contexte où les changements du réseau sont considérés comme des défauts (tolérance aux pannes, algorithmes auto-stabilisants, *etc*.). Au cours de la dernière décennie, la communauté a exploré des contextes où la dynamique est considérée comme une propriété du réseau plutôt qu'une exception. La recherche a mis en évidence l'importance d'étudier et de définir des modèles de mobilité, de caractériser les propriétés temporelles et d'analyser le comportement des algorithmes dans un tel contexte dynamique. Dans un contexte statique, la stabilité permet à un concepteur d'algorithmes d'avoir tous les paramètres pour prévisualiser une exécution sur un réseau donné. Un défi majeur dans les réseaux dynamiques est de prédire la mobilité et de décider si l'évolution de la topologie permet le succès d'un algorithme. De ce point de vue, nous distinguons deux types de réseaux dynamiques : les réseaux dynamiques contrôlés où les contacts et les changements topologiques peuvent être dirigés d'une manière telle qu'ils s'adaptent à l'exécution de l'algorithme; réseaux non contrôlés où l'évolution de la topologie du réseau est totalement indépendante de l'exécution et imprévisible sans aucune analyse. Cette dernière catégorie représente une partie importante des contextes pratiques, tels que les réseaux de véhicules, les réseaux téléphoniques, les réseaux sociaux, *etc*., où les mouvements des unités communicantes dépendent des mouvements des objets mobiles sous-jacents. Les types de dynamique résultant de ces réseaux varient en échelle et en nature. Par exemple, certains de ces réseaux restent connexes tout le temps (O'Dell and Wattenhofer [2005]); d'autres sont toujours déconnectés (Jain *et al.* [2004]) mais offrent une sorte de connexité dans le temps et dans l'espace (connexité temporelle); d'autres sont connexes de manière récurrente, périodiques, *etc.* Tous ces contextes peuvent être représentés sous forme de classes de graphes dynamiques. Une douzaine de classes ont été identifiées et organisées en une hiérarchie (Casteigts *et al.* [2012]).

Les réseaux dynamiques peuvent être modélisés de plusieurs manières. Il est souvent commode, quand on considère la topologie d'un point de vue global (par exemple, une trace enregistrée), d'utiliser des graphes dynamiques représentés comme une suite de graphes statiques $\mathcal{G} = \{G_1, G_2, ..., G_\delta\}$, chaque graphe G_t correspond à l'état du système à un instant t ou pendant un intervalle de temps $[t, t+1)$ (connu sous le nom de graphes évolutifs (Bui-Xuan *et al.* [2003])). Étant donné un graphe dynamique, une question naturelle qui se pose est de savoir à quelles classes il appartient. Cette question est intéressante parce que la plupart des classes

connues de graphes dynamiques correspondent à des conditions nécessaires et/ou suffisantes pour des problèmes ou des algorithmes distribués (diffusion, élection, arbres d'extension, transfert de jetons, *etc.*). Ainsi, être capable de classer un graphe dans la hiérarchie est utile pour déterminer quels problèmes distribués (ou algorithmes) peuvent être résolus (exécutés) avec succès sur ce graphe. De plus, des outils de classification, tels que des algorithmes pour tester l'appartenance de graphes à des classes données et pour tester des propriétés topologiques et temporelles, peuvent être utiles pour choisir un bon algorithme dans des contextes où l'évolution d'un réseau n'est pas connue à l'avance. Un concepteur d'algorithmes peut enregistrer des traces topologiques du monde réel et ensuite tester si les graphes dynamiques correspondants sont inclus dans les classes correspondant aux conditions topologiques nécessaires pour résoudre les problèmes à portée de main. Alternativement, les algorithmes en ligne qui traitent des graphes dynamiques à mesure qu'ils évoluent pourraient atteindre le même objectif sans avoir besoin de recueillir des traces. Dans ce travail, nous apportons une contribution à l'automatisation de la classification des graphes dynamiques. Nous proposons des stratégies pour tester l'appartenance d'un graphe dynamique à une classe donnée et nous définissons un cadre générique pour le test de propriétés dans les graphes dynamiques. Nous explorons également le cas où aucune propriété sur le graphe n'est garantie à travers l'étude du problème de maintien d'une forêt d'arbres recouvrants dans un graphe dynamique.

La première classe de graphes dynamiques que nous étudions est la classe *connexité temporelle*. Dans le chapitre 2, nous abordons le problème de tester si un graphe dynamique donné $\mathcal{G}$ est temporellement connexe, c'est-à-dire un chemin temporel (également appelé un trajet) existe entre toutes les paires de nœuds. En supposant que le réseau est représenté sous la forme d'une séquence $\mathcal{G} = \{G_1, G_2, \ldots, G_\delta\}$, deux variantes du problème sont étudiées, selon que l'on autorise la traversée consécutive d'un seul ou d'un nombre illimité d'arcs à chaque étape (trajets stricts vs non-stricts). Dans le cas des trajets stricts, deux algorithmes pré-existants pour d'autres problèmes peuvent être adaptés. Cependant, nous montrons qu'une approche dédiée permet d'obtenir une meilleure complexité en temps que le premier algorithme dans tous les cas, et que le second dans certaines familles de graphes, notamment les graphes dont la densité est faible à tout instant (bien que potentiellement élevée à travers le temps). La complexité de notre algorithme est en $O(\delta\mu n)$, où δ est le nombre d'étapes $|\mathcal{G}|$ et $\mu = max(|E_t|)$ est le nombre maximal d'arcs pouvant exister à un instant donné. Ce paramètre est à contraster avec $m = |\cup E_t|$, l'union de tous les arcs apparaissant au cours du temps. En effet, il n'est pas rare qu'un scénario de mobilité exhibe à la fois un μ petit et un m grand. Nous caractérisons les principales valeurs charnières de δ, μ et m permettant de décider quel algorithme utiliser. Dans le cas de trajets non-stricts, pour lesquels nous ne connaissons pas d'algorithme existant, nous montrons qu'un prétraitement du graphe d'entrée nous permet de réutiliser le même algorithme qu'auparavant. Par hasard,

ces opérations coûtent à nouveau $O(\delta\mu n)$ temps, ce qui implique que le deuxième problème n'est pas plus difficile que le premier. Nos deux algorithmes construisent graduellement la fermeture transitive des trajets stricts (notée $\mathcal{G}^*_{st}$) ou non-stricts (notée $\mathcal{G}^*$) à mesure que les arcs sont examinés. Ce sont des algorithmes de type *streaming* qui sont aussi capables d'arrêter leur exécution sitôt la connexité temporelle atteinte. Un sous-produit intéressant est de rendre $\mathcal{G}^*_{st}$ et $\mathcal{G}^*$ disponibles pour de futures requêtes d'accessibilité temporelle de type source-destination. Les travaux présentés dans ce chapitre ont été publiés dans les actes d'AlgoTel (Barjon *et al.* [2014c]) et AETOS International Conference (RCFRUS) (Barjon *et al.* [2014b]).

Une autre classe étudiée est la classe *T-intervalle connexité*. Un graphe dynamique est dit T-intervalle connexe si pour tout $t \in [1, \delta - T + 1]$ tous les graphes $\{G_t, G_{t+1}, ..., G_{t+T-1}\}$ partagent un même sous-graphe recouvrant connexe. Cette classe a été identifiée comme jouant un rôle important dans plusieurs problèmes distribués, tels que la détermination de la taille d'un réseau ou le calcul d'une fonction des entrées initiales des nœuds. Nous proposons dans le chapitre 3 une solution au problème de décider si un graphe dynamique donné $\mathcal{G}$ est T-intervalle connexe pour un T donné. Nous considérons également le problème lié de trouver le plus grand T pour lequel un graphe $\mathcal{G}$ donné est T-intervalle connexe. Nous supposons que les changements entre deux graphes consécutifs sont arbitraires et qu'une approche opérant à l'échelle globale de la séquence est donc appropriée. Précisément, nous considérons deux opérations élémentaires qui sont *l'intersection binaire* (étant donné deux graphes, calculer leur intersection) et le *test de connexité* (étant donné un graphe, déterminer s'il est connexe). Nous montrons d'abord que les deux problèmes nécessitent $\Omega(\delta)$ de telles opérations en utilisant l'argument de base que chaque graphe de la séquence doit être considéré au moins une fois. Plus étonnamment, nous montrons que les deux problèmes peuvent être résolus en utilisant seulement $\theta(\delta)$ de telles opérations et nous développons des algorithmes en ligne optimaux qui atteignent cette borne. Par conséquent, le coût des opérations (linéaire en nombre d'arêtes) est contrebalancé par une logique de haut niveau efficace qui pourrait, par exemple, bénéficier de circuits dédiés (ou code optimisé) pour les deux opérations. Nous présentons une deuxième stratégie, donnant des bornes supérieures de $O(\delta \log \delta)$ opérations pour les deux problèmes. Son intérêt principal est dans le fait que ça peut être parallélisé, et ceci nous permet de classer les deux problèmes comme étant dans NC (Nick's class). Les résultats présentés dans ce chapitre ont été publiés dans les actes de la 9ème International Conference on Algorithms and Complexity CIAC (Casteigts *et al.* [2015a]) et AlgoTel (Casteigts *et al.* [2015b]).

Dans le chapitre 4, nous présentons une généralisation du framework présenté dans le chapitre 3 qui permet de tester d'autres classes et propriétés. Suivant le même principe, mais en utilisant des opérations différentes (c'est-à-dire en rem-

plaçant *intersection* et *test de connectivité* par d'autres opérations), d'autres problèmes de classification peuvent être résolus avec la même logique. Nous présentons une solution optimale générale pour trois problèmes différents : nous nous intéressons à la classe $\mathcal{B}$ des graphes dynamiques avec *réapparition bornée dans le temps des arêtes*. Cette classe est définie par une propriété sur la récurrence des arêtes dans le temps dans un graphe dynamique. Un graphe a une réapparition bornée dans le temps des arêtes avec une borne b si le temps entre deux apparences de n'importe quelle arête dans le graphe $\mathcal{G}$ est au plus b. Le problème considéré est celui de trouver la plus petite borne b telle que $\mathcal{G}$ ait une réapparition bornée dans le temps des arêtes. Nous considérons, aussi, le problème de trouver le (pire) *diamètre temporel* d'un graphe dynamique donné $\mathcal{G}$, c'est-à-dire la plus petite durée dans laquelle il existe un trajet depuis n'importe quel nœud vers tous les autres nœuds. Enfin, nous étudions un problème un peu plus complexe, celui de calculer le *diamètre temporel aller-retour* d'un graphe donné $\mathcal{G}$, c'est-à-dire la plus petite durée dans laquelle il y a un trajet aller-retour de n'importe quel nœud vers tous les autres nœuds. Cette approche convient pour une étude de haut-niveau de ces problèmes lorsque les détails des changements entre des graphes successifs dans une séquence sont arbitraires. Si l'évolution du graphe dynamique est limitée à certains égards (par exemple, nombre limité de changements entre graphes), on pourrait tirer profit de l'utilisation de structures de données plus sophistiquées pour réduire la complexité, ce que nous suggérons comme perspective.

Outre la classification, nous sommes également intéressés par l'algorithmique distribué dans les graphes dynamiques. Le chapitre 5 est une tentative de comprendre ce qui peut encore être calculé (et garanti) quand aucune hypothèse n'est faite sur la dynamique du graphe : ni sur le taux de changement, ni sur leur simultanéité, ni sur la connexité globale. En d'autres termes, nous supposons que le graphe n'appartient à aucune des classes présentées. Dans ce contexte chaotique, nous présentons un algorithme qui vise à maintenir le moins possible d'arbres couvrants par composante connexe, tout en garantissant certaines propriétés. Notre algorithme est l'adaptation d'un algorithme haut-niveau (ré-étiquetage de graphe), de Casteigts *et al.* [2013a] et Casteigts [2006], dans un modèle de passage de message synchrone (pour les réseaux dynamiques). Alors que les principes de la variante haut-niveau sont préservés, le nouvel algorithme s'avère beaucoup plus complexe. En particulier, il implique une nouvelle technique qui consiste à maintenir une permutation distribuée de l'ensemble de tous les IDs des nœuds tout au long de l'exécution. L'algorithme hérite également des propriétés de sa variante originale : il repose sur des décisions purement localisées, pour lesquelles aucune information globale n'est jamais collectée par les nœuds, et pourtant il maintient un certain nombre de propriétés critiques quelle que soit la fréquence et l'échelle des changements. En particulier, le graphe reste toujours recouvert par une forêt d'arbres dans laquelle 1) aucun cycle ne peut jamais apparaître, 2) chaque nœud appartient à un arbre, et 3) après un nombre arbitraire de disparition d'arêtes, tous les sous-arbres maximaux

régénèrent exactement et immédiatement un jeton (à leur racine). Ces propriétés sont assurées quelle que soit la dynamique, même si les changement se poursuivent pendant une longue période arbitraire. L'optimalité n'est pas l'objet de cette étude, cependant le nombre d'arbres par composante connexe, la métrique d'intérêt ici, finit par converger à un arbre unique si le réseau cesse de changer (ce qui n'est attendu). Les travaux de ce chapitre ont été publiés dans les actes de la 18ème International Conference on Principles of Distributed Systems OPODIS (Barjon *et al.* [2014a]).

Contents

List of Figures

List of Tables

Introduction

Among the major evolutions in the field of computer science, we note the emergence of dynamic networks. These networks consist of entities making contact over time with one another, which makes them different from static networks where the topology remains unchanged. Earlier, several studies have been made in this discipline to develop new techniques, models, and analyses in order to investigate and solve substantial problems in a context where the network changes are considered as faults (fault-tolerance, self stabilizing algorithms etc.). Over the last decade, the community has explored contexts where the dynamic is considered as a property of the network, rather then exception. Research has highlighted the importance of studying and defining mobility patterns, characterizing dynamic properties, and analysing the behaviour of algorithms in a dynamic context. In a static context, the stability allows an algorithm designer to have all parameters to preview the execution on a given network. A major challenge in dynamic networks is difficulties to predict mobility patterns and decide whether the evolution of the topology satisfies requirements for the success of a given algorithm. From this point of view we distinguish two types of dynamic networks: Controlled dynamic networks where contacts and topological changes can be directed in a way such that they adapt to the execution of the algorithm; non-controlled networks where the evolution of the network topology is completely independent from the execution and unpredictable without any analysis. The latter category represents a significant part of the practical contexts, as vehicular networks, telephone networks, social networks etc. where the movements of communicating and computing units depend on the movements of the underlying mobile objects. The types of dynamics resulting from these networks are varied in scale and nature. For instance, some of these networks remain connected at all times (O'Dell and Wattenhofer [2005]); others are always disconnected (Jain *et al.* [2004]) but still offer some kind of connectivity over time and space (*temporal connectivity*); others are recurrently connected, periodic, etc. All of these contexts can be represented as dynamic graph classes. A dozen such classes were identified and organized into a hierarchy (Casteigts *et al.* [2012]).

Dynamic networks can be modelled in a number of ways. It is often convenient, when looking at the topology from a global standpoint (e.g. a recorded trace), to use dynamic graphs represented as a sequence of static graphs $\mathcal{G} = \{G_1, G_2, ..., G_\delta\}$, each graph G_t corresponds to the state of the network during the time interval

$[t, t+1)$ (known as evolving graphs (Bui-Xuan *et al.* [2003])) or at a time t in the discrete domain (known as untimed evolving graphs). Given a dynamic graph, a natural question to ask is to which of the classes this graph belongs. This question is interesting because most of the known classes of dynamic graphs correspond to necessary or sufficient conditions for given distributed problems or algorithms (broadcast, election, spanning trees, token forwarding, etc.). Thus, being able to classify a graph in the hierarchy is useful for determining which distributed problems (or algorithms) can be successfully solved (executed) on that graph. Furthermore, classification tools, such as algorithms for testing the membership to given classes and for testing properties, can be useful for choosing a good algorithm in settings where the evolution of a network is not known in advance. An algorithm designer can record topological traces from the real world and then test whether the corresponding dynamic graphs are included in classes that correspond to the topological conditions for the problem at hand. Alternatively, online algorithms that process dynamic graphs as they evolve could accomplish the same goal without the need to collect traces. In this work we provide a contribution to the automation of dynamic graphs classification. We provide strategies for testing the membership of a dynamic graph to a given class and a generic framework to test properties in dynamic graphs. We also explore the case where no property on the graph is guaranteed through the problem of maintaining a spanning forest in highly dynamic graphs.

The first dynamic graph class that we study is the class *temporal connectivity*. In Chapter 2, we address the problem of testing whether a given dynamic graph $\mathcal{G}$ is temporally connected, i.e. a *temporal path* (also called a *journey*) exists between all pairs of nodes. Assuming the network is represented as a sequence $\{G_1, G_2, ..., G_\delta\}$, two cases are studied depending on whether a single edge or an unlimited number of edges can be crossed in a same G_t (respectively, strict journeys vs non-strict journeys). In the case of strict journeys, a number of existing algorithms designed for more general problems can be adapted. We adapt one of them to the above formulation of the problem and characterize its running time complexity. The parameters of interest are the length of the graph sequence δ, the maximum instant density $\mu = max(|E_i|)$, and the cumulated density $m = |\cup E_i|$. Our algorithm has a time complexity of $O(\delta\mu n)$, where n is the number of nodes. This complexity is compared to that of the other solutions: one is always more costly (although it solves a more general problem), the other one is more or less costly depending on the interplay between instant density and cumulated density. Our solution is relevant for sparse mobility scenarios (e.g. robots or UAVs exploring an area) where the number of neighbors at a given time is low, though many nodes can be seen over the whole execution. In the case of non-strict journeys, for which no algorithm is known, we show that some pre-processing of the input graph allows us to re-use the same algorithm than before. By chance, these operations happen to cost again $O(\delta\mu n)$ time, which implies that the second problem is not more difficult than the

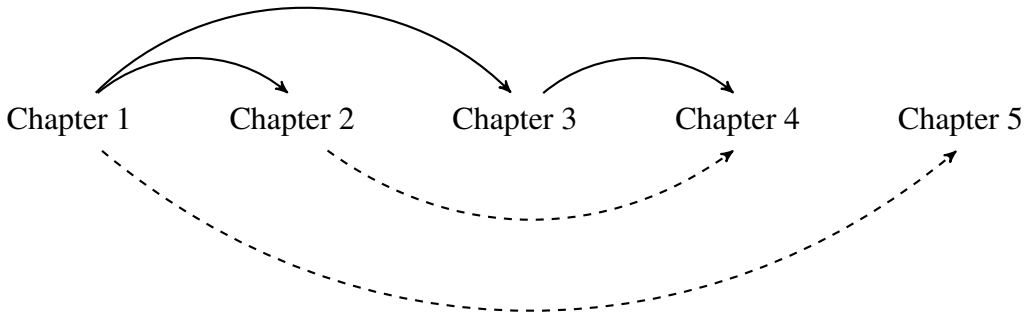
first. Both algorithms gradually build the transitive closure of strict journeys or non-strict journeys as the edges are examined; these are streaming algorithms. They stop their execution whenever temporal connectivity is satisfied (or after the whole graph has been examined). A by-product of the execution is to make the transitive closure available for further connectivity queries (in a temporal version), these queries being then reduced to simple adjacency tests in a static graph. The work presented in this chapter has been published in AlgoTel (Barjon *et al.* [2014c]) and AETOS International Conference (RCFRUS) (Barjon *et al.* [2014b]).

Another studied class is *T-interval connectivity*. A dynamic graph is said to be T-interval connected if for all $t \in [1, \delta - T + 1]$ all graphs $\{G_t, G_{t+1}, ..., G_{t+T-1}\}$ share a common connected spanning subgraph. The class of T-interval connected graphs was identified as playing an important role in several distributed problems, such as determining the size of a network or computing a function of the initial inputs of the nodes. We propose in Chapter 3 a solution to the problem of deciding whether a given dynamic graph $\mathcal{G}$ is T-interval connected for a given T. We also consider the related problem of finding the largest T for which the given $\mathcal{G}$ is T-interval connected. We assume that the changes between two consecutive graphs are arbitrary and that no specific data structure is used to represent the sequence of graphs. As such, we focus on high-level strategies that work directly at the level of the graph sequence. Precisely, we consider two graph-level operations as building blocks, which are *binary intersection* (given two graphs, compute their intersection) and *connectivity testing* (given a graph, test if it is connected). Put together, these operations have a strong and natural connection with the problems at stake. We first show that both problems require $\Omega(\delta)$ such operations using the basic argument that every graph of the sequence must be considered at least once. More surprisingly, we show that both problems can be solved using only $\Theta(\delta)$ such operations and we develop optimal online algorithms that achieve these matching bounds. Hence, the cost of the operations – both linear in the number of edges – is counterbalanced by efficient high-level logic that could, for instance, benefit from dedicated circuits (or optimized code) for both operations. We present a second strategy, yielding upper bounds of $O(\delta \log \delta)$ operations for both problems. Its main interest is in the fact that it can be parallelized, and this allows us to classify both problems as being in NC (i.e. Nick's class). The results presented in this chapter were published in the 9th International Conference on Algorithms and Complexity CIAC (Casteigts *et al.* [2015a]) and AlgoTel (Casteigts *et al.* [2015b]).

In Chapter 4 we present a generalization of the framework presented in Chapter 3 that allows one to test other classes and properties. Following the same principle, but using different operations (i.e. replacing *intersection* and *connectivity test* by other operations), other classification problems can be solved with the same high-level logic. We present a general optimal solution for three different problems: We are interested in the Class $\mathcal{B}$ of those dynamic graphs with *Time-bounded reappear-*

ance of edges. This class is defined by a property on edges recurrence over time in a dynamic graph. A graph has a time-bounded edges reappearance with a bound b if the time between two appearances of the same edge in the graph $\mathcal{G}$ is at most b. The considered problem is that of finding the smallest bound b such that $\mathcal{G}$ has a time-bounded edges reappearance. Then, we consider the problem of finding the (worst) *temporal diameter* of a given dynamic graph $\mathcal{G}$ i.e. the smallest duration in which there exist a journey from any node to all other nodes. Finally, we investigate a somewhat more complex problem, that of computing the *round trip diameter* of a given graph $\mathcal{G}$ i.e. the smallest duration in which there exist a back-and-forth journey from any node to all other nodes. This approach is suitable for a high-level study of these problems when the details of changes between successive graphs in a sequence are arbitrary. If the evolution of the dynamic graph is constrained in some ways (e.g., bounded number of changes between graphs), then one could benefit from the use of more sophisticated data structures to lower the complexity of the problem, which we suggest as a perspective. The work in this chapter is the most recent in the thesis and has not yet been published.

Besides the classification, we are also interested in *distributed* algorithms in dynamic graphs. Chapter 5 is an attempt to understand what can still be computed (and guaranteed) when no assumptions are made on the graph dynamics: neither on the rate of change, nor on their simultaneity, nor on global connectivity. In other words, we do not assume that the graph belongs to any of presented classes. In this seemingly chaotic context, we present an algorithm that strives to maintain as few trees per components as possible, while always guaranteeing some properties. Our algorithm is the adaptation of a coarse-grain interaction algorithm, from Casteigts *et al.* [2013a] and Casteigts [2006], into the synchronous message passing model (for dynamic networks). While the high-level principles of the coarse-grain variant are preserved, the new algorithm turns out to be significantly more complex. In particular, it involves a new technique that consists of maintaining a distributed permutation of the set of all nodes IDs throughout the execution. The algorithm also inherits the properties of its original variant: It relies on purely localized decisions, for which no global information is ever collected at the nodes, and yet it maintains a number of critical properties whatever the frequency and scale of the changes. In particular, the graph remains always covered by a spanning forest in which 1) no cycle can ever appear, 2) every node belongs to a tree, and 3) after an arbitrary number of edge disappearance, all maximal sub-trees immediately restore exactly one token (at their root). These properties are ensured whatever the dynamics, even if it keeps going for an arbitrary long period of time. Optimality is not the focus here, however the number of tree per components – the metric of interest here – eventually converges to one if the network stops changing (which is never expected to happen,though). The work in this chapter has been published in the 18th International Conference on Principles of Distributed Systems OPODIS (Barjon *et al.* [2014a]).



Dependencies between the chapters of this thesis.
Dashed links represents weak dependencies.

Chapter 1

Background on Dynamic Graphs and Classification Problems

Contents

In this chapter we give definitions of general concepts and notations that will be used in this document. In Section 1.1, we present different representations of dynamic graphs and we define related concepts. In Section 1.2, we define some important properties and hierarchy of dynamic graph classes. Finally, Section 1.3 presents the motivation of our work and introduces some of the problems we will address on dynamic graph classification.

1.1 Dynamic Graphs

In this section we present various representations of dynamic graphs and some basic notions and concepts that will be used in this thesis.

1.1.1 Dynamic Graphs Models

In a dynamic context, networks can be represented in many ways, it depends on the definition of the temporal domain in which the system evolves. We review below some of the models used in the literature.

Time-Varying Graphs: Let's first start with the most general case (continuous time, arbitrary dynamics):

Definition 1.1 (Time-varying Graph Casteigts *et al.* [2012])**.** *A Time-Varying Graph denoted TVG is a quintuplet $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$ where:*

- *V is a static set of nodes and ;*
- *E is the set of possible edges;*
- *$\mathcal{T} \subseteq \mathbb{T}$ is the life time of the system (the time interval wherein the system exists), the temporal domain $\mathbb{T}$ is generally assumed to be $\mathbb{R}^+$ for continuous-time systems or $\mathbb{N}$ for discrete-time systems;*
- *$\rho : E \times \mathcal{T} \to \{0, 1\}$ is a function that determines the presence of a given edge at a given time.*
- *$\zeta : E \times \mathcal{T} \to \mathbb{T}$ is a function that determines the latency (the time a message takes to cross the edge) of a given edge at a given time.*

This model can be extended further by adding a node presence function $\psi : V \times \mathcal{T} \to \{0, 1\}$ *(i.e., the presence of nodes varies) or a* node latency function $\varphi : V \times \mathcal{T} \to \mathbb{T}$ *(accounting e.g. for local processing times at routers), etc.*

Notation $\mathcal{G}_{[i,j]}$ denotes the subgraph of $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$ induced by the period $[i, j] \subseteq \mathcal{T}$, i.e. $\mathcal{G}_{[i,j]} = (V, E, [i, j], \rho, \zeta)$.

Figure 1.1 shows an example of a TVG where labels on the edges indicate the time intervals in which these edges are present (according to the function ρ).

Definition 1.2 (The Footprint of a dynamic graph)**.** *We define the footprint of the dynamic graph $\mathcal{G}$ by the simple graph $G = (V, E)$ where V is the static set of nodes in the dynamic graph and E is the set of all edges that exist at least for a time in the*

life time of the graph. This concept is also referred to as underlying graph in the literature (except that the underlying graph may contain edges that do not appear).

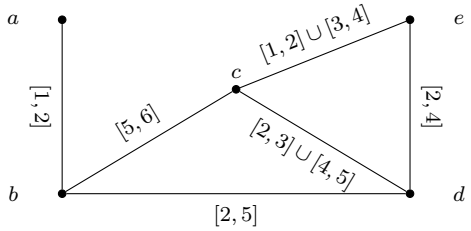


Figure 1.1: Example of a TVG.

Evolving graphs: It is often convenient to look at the evolution of the network as a sequence of global snapshots. One such model is called evolving graphs (Ferreira [2002]). Two variants exist, depending on the way time is associated with the sequence. In the most general case, the evolution of the graph is modelled by an indexed sequence of graphs $\{G_i = (V, E_i)\}$, each graph G_i being associated to a time interval $[t_i, t_{i+1})$ during which it is available. Formally:

Definition 1.3 (Evolving Graph)**.** *Let $G = (V, E)$ be a graph, an* evolving graph $\mathcal{G}$ *is represented by a triplet $\mathcal{G} = (G, \mathcal{S}_G, \mathcal{S}_\mathbb{T})$ where:*

- *$\mathcal{S}_G$ is an indexed sequence $\{G_i = (V, E_i)\}$ of G subgraphs;*
- *$\mathcal{S}_\mathbb{T} \subseteq \mathbb{T} \subseteq \mathbb{N}$ or $\mathbb{R}$ is the associated sequence of dates $t_1, t_2, ...$ where $\forall e \in E_i$, e is present in the time interval $[t_i, t_{i+1})$ (only the graph G_i is present in this interval).*

From a mathematical point of view, evolving graphs are almost equivalent to time-varying graphs. The only difference is that the topological events are required to be countable (due to the representation as a sequence), whereas they can be arbitrary with time-varying graphs. This leads to strong differences in the expressivity of both models and thus to the potential strength of an adversary in distributed networks (Casteigts *et al.* [2013b]). From a usage point of view, the main difference is in the convenience of the formalism (notations), which makes evolving graph often useful in a discrete setting, while TVGs are particularly useful to describe distributed algorithms in a continuous-time settings (or, as we will see, to express general properties on the network dynamics).

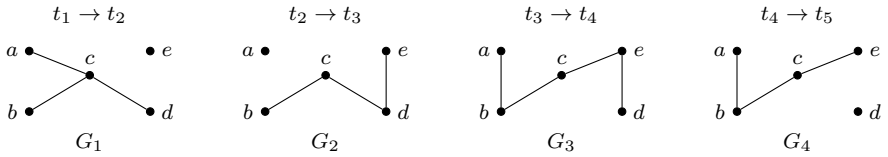


Figure 1.2: Example representation of a dynamic graph as an evolving graph.

We show in Figure 1.2 a representation of an evolving graph $\mathcal{G}$ in the form of a sequence based representation corresponding to the sequences $\mathcal{S}_G$ and $\mathcal{S}_{\mathbb{T}}$. Note that evolving graphs can also be represented in a compact way like the one of Figure 1.1.

To simplify the model and adapt it to our problems (most of which are in discrete time), we will use throughout this document a lightened model called *untimed* evolving graphs where the durations of the periods are ignored. A dynamic graph in the following is represented by a mere sequence of indexed graphs whose only the order describes the evolution of the graph:

Definition 1.4 (Untimed Evolving Graph)**.** *An* untimed evolving graph *is an indexed sequence $\{G_i = (V, E_i)\}$ where V is a static set of nodes and $E_i \subseteq E$ is a dynamically changing set of edges.*

We use in the following chapters:

- *$\{G_1, G_2, ..., G_\delta\}$ to denote a finite untimed evolving graph of length δ;*
- *n to denote the number of nodes $|V|$;*
- *$\delta = |\mathcal{G}|$ to denote the length of the graph i.e. the number of graphs in $\mathcal{G}$;*
- *$\{G_1, G_2, ...\}$ to denote an infinite untimed evolving graph.*

We consider in a general case that there is no assumptions and no restrictions on the set of edges changing function (changes are arbitrary), if there is one in a later context, this will be specified.

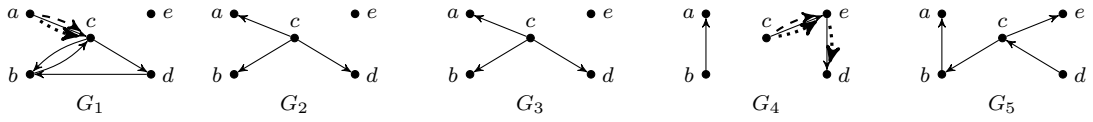


Figure 1.3: Example representation of a dynamic graph as an untimed evolving graph.

Figure 1.3 shows a representation of an untimed evolving graph $\mathcal{G} = \{G_1, G_2, \ldots, G_5\}$ of length $\delta = 5$. We consider the general case where edges are directed.

Other models that are widely used exist in the literature, like *temporal networks* Kempe *et al.* [2000] *link streams* Viard *et al.* [2016].

1.1.2 Basic Definitions on Dynamic Graphs

An important property in dynamic graphs is the connectivity between nodes. Let us first look at this property in the context of untimed evolving graphs. In Figure 1.3, at no time, the graph $\mathcal{G}$ is connected in a classical sense. In other words, no graph in $\{G_i : 1 \geq i \geq 5\}$ is connected. We can note that at any time i, a graph G_i allows only to a limited number of nodes to be connected (with a path). For instance, at no time i, a is connected to e. But by allowing the combination of edges from different graphs, e can be reached from a by means of a temporal path, e.g: $((a, c) \in E_1, (c, e) \in E_4)$. This temporal aspect of a path gives the notion of *journey* (temporal path). If we consider the possibility of crossing only one edge per graph, a could reach e as shown above but it could not reach d. While if it is allowed to cross several edges per graph, a can use the journey $((a, c) \in E_1, (c, e) \in E_4, (e, d) \in E_4)$ to reach d. This restriction gives two types of journeys: in the first case we talk about *strict journey* represented by dashed arrows in Figure 1.3, in the second case, with the relaxation of the constraint we used a *non-strict journey* (or *journey* in general) represented by dotted arrows in Figure 1.3. More formally, it is defined as follows:

Definition 1.5 (Non-strict Journey)**.** *$\mathcal{J}(u, v) = \{(e_1, t_1), (e_2, t_2), \ldots, (e_p, t_p)\}$ ($\mathcal{J}$ when the context is implicit), such that $e_i \in E_{t_i}$, is a* non-strict journey *from u to v in $\mathcal{G}$ if and only if $e_1, e_2, ..., e_p$ is a path from u to v and for all $i \in 1..p-1$, $t_{i+1} \geq t_i$. The existence of a non-strict journey from u to v, when the context is implicit, is noted $u \rightsquigarrow v$.*

Definition 1.6 (Strict Journey)**.** *$\mathcal{J}^{st}(u, v) = \{(e_1, t_1), (e_2, t_2), \ldots, (e_p, t_p)\}$ ($\mathcal{J}^{st}$ when the context is implicit), such that $e_i \in E_{t_i}$, is a* strict journey *from u to v in $\mathcal{G}$ if and only if $e_1, e_2, ..., e_p$ is a path from u to v and for all $i \in 1..p-1$, $t_{i+1} > t_i$. Note that the inequality $t_{i+1} > t_i$ is strict, i.e. at most one edge can be crossed in a single step t_i (as opposed to an unlimited number for non-strict journeys). The existence of a strict journey from u to v, when the context is implicit, is noted $u \overset{st}{\rightsquigarrow} v$.*

In the general case of TVG (time varying graph) we define a journey as follows:

Definition 1.7 (Journey in TVG). $\mathcal{J}(u,v) = \{(e_1,t_1),(e_2,t_2),\ldots,(e_p,t_p)\}$ *($\mathcal{J}$ when the context is implicit) is a* journey *from u to v in a TVG $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$ if and only if $e_1, e_2, ..., e_p$ is a path from u to v and for all $i \in 1..p-1$, $\forall t \in [t_i, \zeta(e_i,t_i)]$, $\rho(e_i,t) = 1$ and $t_{i+1} \geq t_i + \zeta(e_i,t_i)$.*

The existence of a journey from u to v, when the context is implicit, is noted $u \rightsquigarrow v$.

For example, in Figure 1.1, with $\zeta \leq 1$, $\{((a,b),1),((b,d),2),((d,e),3)\}$ is a journey from a to e. There is no journey from a to e that uses the sequence of edges $(a,b),(b,c),(c,d)$, because (c,d) appears always before any appearance of (b,c).

Note that the temporal nature of the dynamic graph gives an orientation to the journey even if the graph G is not directed, i.e the existence of a journey from a to e does not implies the existence of a journey from e to a.

Definition 1.8 (Departure of a journey). *The departure $departure(\mathcal{J}(u,v))$ of a journey $\mathcal{J}(u,v) = \{(e_1,t_1),(e_2,t_2),\ldots,(e_p,t_p)\}$ is the date t_1.*

Definition 1.9 (Arrival of a journey). *The arrival $arrival(\mathcal{J}(u,v))$ of a journey $\mathcal{J}(u,v) = \{(e_1,t_1),(e_2,t_2),\ldots,(e_p,t_p)\}$ is the date $t_p+\zeta(e_p,t_p)$ ($arrival(\mathcal{J}(u,v)) = t_p$ in untimed evolving graph model where communication latency on existing edges is ignored).*

In general (whatever the model used), we denote the set of all possible journeys in dynamic graph $\mathcal{G}$ as $\mathcal{J}^*(\mathcal{G})$ and the set of all possible journeys in a period $[i,j]$ as $\mathcal{J}^*(\mathcal{G},i,j)$ ($\mathcal{J}^*(i,j)$ when the context is implicit). Similarly $\mathcal{J}^*(u,v,\mathcal{G})$ denotes the set of possible journeys from u to v in all the graph $\mathcal{G}$ and $\mathcal{J}^*(u,v,\mathcal{G},i,j)$ ($\mathcal{J}^*(u,v,i,j)$ when the context is implicit) denotes the set of possible journeys from u to v in the graph $\mathcal{G}$ in a period $[i,j]$.

As a path in a static graph, a journey $\mathcal{J}(u,v)$ in a dynamic graph has a topological length which corresponds to the number of hops in it, i.e. p for a journey $\{(e_1,t_1),(e_2,t_2),\ldots,(e_p,t_p)\}$. It also has a temporal length (or duration), which is the time a message takes to reach v departing from u i.e. $arrival(\mathcal{J}(u,v)) - departure(\mathcal{J}(u,v))$. This notion gives the concept of temporal distance in a graph:

Definition 1.10 (Temporal distance at time t). *Let $\mathcal{J}^*(u,v,\mathcal{G},t)$ be the set of journeys from u to v whose $departure \geq t$. We define the temporal distance $tDis(u,v,\mathcal{G},t)$ from u to v at time t by the duration $min\{arrival(\mathcal{J}(u,v)) - t : \mathcal{J}(u,v) \in \mathcal{J}^*(u,v,\mathcal{G},t)\}$.*

From this, we define the concept of *temporal diameter*. Informally, the temporal diameter of a graph at time t (step in untimed evolving graphs) is the largest temporal distance between any pair of nodes at this time. Formally:

Definition 1.11 (Temporal diameter at time t)**.** *The temporal diameter $tDiam(\mathcal{G}, t)$ of a dynamic graph $\mathcal{G}$ at time t is the largest temporal distance between any pair of nodes at this time: $max\{tDis(u, v, \mathcal{G}, t) : u, v \in V\}$.*

Definition 1.12 (Temporal diameter of a dynamic graph $\mathcal{G}$)**.** *The temporal diameter $tmpDiam(\mathcal{G})$ of a dynamic graph $\mathcal{G}$ is the largest temporal diameter $tDiam(\mathcal{G}, t)$ at any time t: $max\{tDiam(\mathcal{G}, t) : t \in \mathcal{T}\}$.*

1.2 Dynamic Graph Classes

In this section we define some classes of dynamic graphs based on topological and temporal properties. Then, we introduce the main topic of this work, which is the automatic classification of dynamic graphs.

1.2.1 Temporal Properties and Dynamic Graph Classes

After defining the most important concepts, we are interested in topological and temporal properties in dynamic graphs. In our context, a question is what gives sense to a given property and makes it interesting to consider. An efficient approach is to analyze distributed algorithms and to look at the assumptions made on the dynamic graph.

Characterizing temporal properties

Casteigts *et al.* [2009] propose a framework that allows to examine what impact a property has on the execution of a distributed algorithm. Their general methodology consists in considering a problem, then characterizing the necessary and/or sufficient conditions for the success of its algorithm execution in terms of network dynamics. The framework is based on the combination of *local computations* by means of *graph relabelings* (Litovsky *et al.* [1999]), and *evolving graphs* as formalism for dynamic networks.

They model this in two predicates: *objective* and *condition*. The objective $\mathcal{O}_{\mathcal{A}}$ defines the success of the execution of an algorithm $\mathcal{A}$. Precisely the system configuration we want to achieve at the end of the execution formally expressed by

a terminal state. In this case we talk about a type of algorithms whose execution ends with the achievement of a precise desired state. Another type of algorithms are executed in order to maintain a correct system configuration during the execution (infinite execution in some cases). The corresponding objective should describe, in this case, the desired state of the system for a series of steps in the execution corresponding to a sequence of graphs in the dynamic graph.

The conditions are represented by *temporal and topological properties* of the dynamic graph $\mathcal{G}$. There are two types of conditions that can be put in relation with the objective of an algorithm: i) *Necessary condition* $\mathcal{C}_{\mathcal{N}}$, without which the objective can not be reached. ii) *Sufficient condition* $\mathcal{C}_{\mathcal{S}}$, which guarantees the objective (given some additional assumptions on the interaction scheduler). Formally:

i) $\forall \mathcal{G}, \neg \mathcal{C}_{\mathcal{N}}(\mathcal{G}) \Rightarrow \neg \mathcal{O}_{\mathcal{A}}(\mathcal{G})$

ii) $\forall \mathcal{G}, \mathcal{C}_{\mathcal{S}}(\mathcal{G}) \Rightarrow \mathcal{O}_{\mathcal{A}}(\mathcal{G})$

For example, the distinction between a strict journey and a non-strict journey was introduced in Casteigts *et al.* [2009] to report on necessary or sufficient conditions on the dynamics of the graph, regarding distributed algorithms based on pairwise interactions. The analysed algorithm $\mathcal{A}$ is a propagation algorithm, which consists in transmitting an information I from a node u to all the other nodes of the graph. In this instance, the objective $\mathcal{O}_{\mathcal{A}}$ that the execution must reach is the final state where $\forall v \in V, v$ has the information I.

We define two properties based on the notion of journey:

Property 1 There exists a journey from a node u to all other nodes of the graph: $\exists u \in V : \forall v \in V, u \rightsquigarrow v$.

Property 2 There exists a strict journey from a node u to all other nodes of the graph: $\exists u \in V : \forall v \in V, u \overset{st}{\rightsquigarrow} v$.

As explained in Casteigts *et al.* [2009], *Property 1*, with $u =$ *the initial emitter*, is a necessary condition so that the algorithm $\mathcal{A}$ can achieve the objective $\mathcal{O}_{\mathcal{A}}$. The proof is based on the argument that the existence of a journey $u \rightsquigarrow v$ is necessary in order to transmit information from u to v. With the assumption $(a1)$ that the transition $G_i \rightarrow G_{i+1}$ allows time to each node that has the information at time i to transmit it to all its neighbors at least once, it was as well shown that *Property 2* is a sufficient condition. *Property 1* is not sufficient because in the assumption $(a1)$ the transmission is guaranteed only for nodes having the information at time i

but a non-strict journey can make multiple hops in the same graph. So nodes that receive the information during the interval $(i, i+1)$ do not necessarily have the time to transmit it in turn to all their neighbors.

Note that the conditions discussed here are tight. However, this is not always the case. The tightness of necessary or sufficient conditions in this particular framework is discussed in Marchand de Kerchove and Guinand [2012] .

Dynamic graph classes

We present here some of the dynamic graph classes introduced in Casteigts *et al.* [2009] and Casteigts *et al.* [2012]. The numbers and names of classes used here might be different from those in these articles, which are themselves different from one another.

From *Property 1* and *Property 2* we define the following two classes of dynamic graphs:

Definition 1.13 (Source Connectivity, *Class 1*)**.** *A dynamic graph $\mathcal{G}$ is in the class* Source Connectivity *iff there exists a journey from a node u to all other nodes of the graph:* $\exists u \in V, \forall v \in V, u \rightsquigarrow v$.

Definition 1.14 (Strict Source Connectivity, *Class 2*)**.** *A dynamic graph $\mathcal{G}$ is in the class* Strict Source Connectivity *iff there exists a strict journey from a node u to all other nodes of the graph:* $\exists u \in V, \forall v \in V, u \overset{st}{\rightsquigarrow} v$.

An other class based on the concept of journey is the class *Sink Connectivity*:

Definition 1.15 (Sink Connectivity, *Class 3*)**.** *A dynamic graph $\mathcal{G}$ is in the class* Sink Connectivity *iff there exists a journey to a node u from all other nodes of the graph:* $\exists u \in V, \forall v \in V, v \rightsquigarrow u$.

In a dynamic graph, this condition is necessary for collecting information by a node from the other nodes. This allows for example a node to compute a function whose inputs are distributed on the other nodes. Or solve the counting problem in the graph.

A class that is included in the three classes presented above is that of *Temporal Connected graphs*:

Definition 1.16 (Temporal Connectivity, *Class 4*)**.** *A dynamic graph $\mathcal{G}$ is* temporally connected *iff for all pairs of nodes $u, v \in V$, there exist a journey $u \rightsquigarrow v$: $\forall u, v \in V, u \rightsquigarrow v$.*

Definition 1.17 (Strict Temporal Connectivity, *Class 5*)**.** *In the same way, a dynamic graph $\mathcal{G}$ is* strictly temporally connected *iff for all pairs of nodes $u, v \in V$, there exist a strict journey $u \overset{st}{\rightsquigarrow} v$: $\forall u, v \in V, u \overset{st}{\rightsquigarrow} v$.*

The class *Temporal Connectivity* defines the set of graphs in which any node can perform a broadcast to all other nodes, and can collect information from all the other nodes.

Definition 1.18 (Round Trip Temporal Connectivity, *Class 6*)**.** *A dynamic graph $\mathcal{G}$ is* round trip temporally connected *iff for all pairs of nodes $u, v \in V$, there exist a journey $\mathcal{J}(u, v)$ and a journey $\mathcal{J}(v, u)$ such that $departure(\mathcal{J}(v, u)) \geq arrival(\mathcal{J}(u, v))$. Formally, $\forall u, v \in V$, $\exists \mathcal{J} \in \mathcal{J}^*(u, v)$,$\exists \mathcal{J} \in \mathcal{J}^*(v, u)$, $departure(\mathcal{J}) \geq arrival(\mathcal{J})$.*

This condition is necessary, for example, for the success of different distributed algorithms requiring a termination detection.

Definition 1.19 (Recurrent Temporal Connectivity, *Class 7*)**.** *A dynamic graph $\mathcal{G}$ has a* Recurrent Temporal Connectivity *iff any subgraph $\mathcal{G}_{[t,+\infty)}$ of the graph $\mathcal{G}$ is temporally connected: $\forall u, v \in V, \forall t \in \mathcal{T}, \exists \mathcal{J} \in \mathcal{J}^*(u, v), departure(\mathcal{J}) > t$.*

This condition is often implicitly assumed in the context of delay tolerant networks.

So far we have presented classes whose definitions are based on the concept of journey. Other classes are defined by properties on direct contacts between nodes:

Definition 1.20 (Temporal star, *Class 8*)**.** *A dynamic graph $\mathcal{G}$ is a* temporal star *iff the footprint G has a spanning subgraph $G' \subseteq G$ that is a star: $\exists u \in V, \forall v \in V, (u, v) \in E(G)$.*

Definition 1.21 (Temporal Completeness, *Class 9*)**.** *A dynamic graph $\mathcal{G}$ is* temporally complete *iff the footprint G is complete: $\forall u, v \in V, (u, v) \in E(G)$.*

Definition 1.22 (Recurrent Temporal Completeness, *Class 10*)**.** *A dynamic graph $\mathcal{G}$ has a* Recurrent Temporal Completeness *iff any subgraph $\mathcal{G}_{[t,+\infty)}$ of the graph $\mathcal{G}$ is temporally complete: $\forall u, v \in V, \forall t \in \mathcal{T}, \exists t' > t, \rho((u, v), t') = 1$.*

Now we are interested in the instant-connectivity of the graph and its evolution over time. The four following classes make sense only in the sequence based model (untimed evolving graphs)

Definition 1.23 (Constant Connectivity, *Class 11*)**.** *A dynamic graph* $\mathcal{G} = \{G_i = (V, E_i)\}$ *is* constantly connected *iff all the graphs* $\{G_i = (V, E_i)\}$ *are connected:* $\forall G_i \in \mathcal{G}, G_i$ *is connected.*

Definition 1.24 (T-interval Connectivity, *Class 12*)**.** *A dynamic graph* $\mathcal{G} = \{G_i = (V, E_i)\}$ *is* T-interval connected *iff every sequence of length* T *in the dynamic graph* $\mathcal{G}$ *shares a common connected spanning subgraph:* $\forall t \in [1, \delta - T + 1], \exists G' \subseteq G, V(G') = V(G), G'$ *is connected, and* $\forall i' \in [t, t + T - 1], G' \subseteq G_{i'}$.

Definition 1.25 (Recurrent Instant-Connectivity, *Class 13*)**.** *An infinite dynamic graph* $\mathcal{G} = \{G_1, G_2, ...\}$ *has a* Recurrent instant-Connectivity *iff there is always a future step* i *such that* G_i *is connected:* $\forall t \in \mathbb{N}, \exists t' \in \mathbb{N} t' \geq t, G_{t'}$ *is connected.*

Definition 1.26 (Recurrent Instant-Routability, *Class 14*)**.** *An infinite dynamic graph* $\mathcal{G} = \{G_1, G_2, ...\}$ *has a* Recurrent instant-Routability *iff for any pair of nodes there is always a future step* i *such that a path exists between these two nodes in the graph* G_i*:* $\forall u, v \in V, \forall t \in \mathbb{N}, \exists t' \in \mathbb{N}, t' \geq t$*, a path exists from* u *to* v *in* $G_{t'}$.

The dynamics of the graph is characterized by the permanent change of the edges set. It would therefore be interesting to look at the classes defined by patterns on the reappearance of the edges:

Definition 1.27 (Recurrence of Edges, *Class 15* or *Class* $\mathcal{R}$)**.** *A dynamic graph* $\mathcal{G}$ *has a* Reappearance of Edges *iff any edge in the footprint reappears infinity often:* $\forall e \in E(G), \forall t \in \mathcal{T}, \exists t' > t, \rho(e, t') = 1$.

Definition 1.28 (Time-bounded Reappearance of Edges, *Class 16* or *Class* $\mathcal{B}$)**.** *A dynamic graph* $\mathcal{G}$ *has a* Time-bounded Reappearance of Edges *with bound* b *iff the waiting time between two appearances of the same edge in the graph* $\mathcal{G}$ *is less than* b *and* G *is connected:* $\forall e \in E(G), \forall t \in \mathcal{T}, \exists t' \in [t, t + b), \rho(e, t') = 1$*, for some* $b \in \mathbb{T}$ *and* G *is connected.*

Definition 1.29 (Periodic Reappearance of Edges, *Class 17* or *class* $\mathcal{P}$)**.** *A dynamic graph* $\mathcal{G}$ *has a* Periodic Reappearance of Edges *with period* p *iff the number of steps*

between two consecutive appearances of the same edge in the graph $\mathcal{G}$ is p and G is connected: $\forall e \in E(G), \forall t \in \mathcal{T}, \forall k \in \mathbb{N}, \rho(e,t) = \rho(e,t+kp)$, *for some* $p \in \mathbb{T}$ *and G is connected.*

Figure 1.4 shows the inclusion relationship between classes.

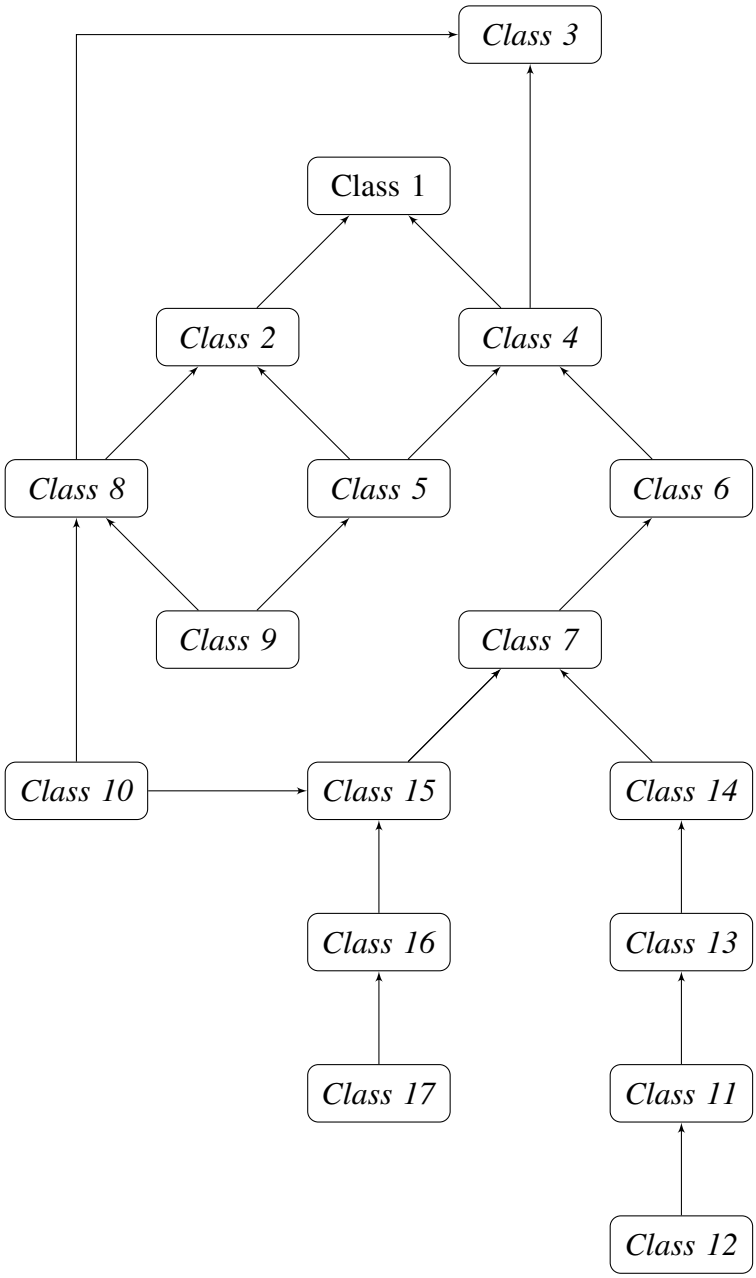


Figure 1.4: Dynamic graph classes hierarchy.

Problem comparison using the dynamic graph hierarchy

A hierarchy of this type can be useful in several studies and formal analyses, in particular, the possibility to transpose results or to compare solutions or problems.

Casteigts *et al.* [2014] are interested in the last three classes $\mathcal{R}$, $\mathcal{B}$, and $\mathcal{P}$. They focus on the problem of reaching from a source node the other nodes of a dynamic graph. The proposed solution and analysis apply to dynamic networks represented as a TVG $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$. More precisely the authors address the problem of broadcast with termination detection, denoted TDB, based on the concept of *journey* and that according to three optimal modes detailed in Bui-Xuan *et al.* [2003]:

- **Foremost:** The messages reach the destinations the earliest possible. For example from the source node a in Figure 1.5, the journeys taken for a foremost broadcast are $\mathcal{J}(a,b) = \{((a,b),1)\}$, $\mathcal{J}(a,c) = \{((a,b),1),((b,c),1+\zeta)\}$ and $\mathcal{J}(a,d) = \{((a,b),1),((b,c),1+\zeta),((c,d),5)\}$. With edges latency $\zeta \leq 1$.

- **Shortest:** The messages take journeys of minimum hop length to reach the destinations ($\mathcal{J}(a,b) = \{((a,b),1)\}$, $\mathcal{J}(a,c) = \{((a,c),2)\}$, $\mathcal{J}(a,d) = \{((a,c),2),((c,d),5)\}$, Figure 1.5).

- **Fastest:** The period of transit of messages to reach the destinations is minimized ($\mathcal{J}(a,b) = \{((a,b),2)\}$, $\mathcal{J}(a,c) = \{((a,c),2)\}$, $\mathcal{J}(a,d) = \{((a,c),5-\zeta)$
$,((c,d),5)\}$, Figure 1.5).

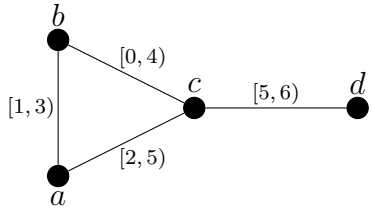


Figure 1.5: A TVG $\mathcal{G}$; the labels on the edges indicate the time intervals in which those edges are present. The edge latency is $\zeta \leq 1$.

In Casteigts *et al.* [2014], an analysis of the computational relationship between the three classes ($\mathcal{R}$, $\mathcal{B}$, and $\mathcal{P}$) was presented by examining the feasibility of the

TDB problem with different levels of knowledge and the reusability of the resulting broadcast trees in these three classes of graphs.

They proved that TDB is not feasible in any of the three classes of graphs without any knowledge on the graph. The minimum knowledge required is n the number of nodes, which allows us to make a *foremost TDB* in Class $\mathcal{R}$ in an unbounded time, in $O(n.b)$ in Class $\mathcal{B}$, and also in Class $\mathcal{P}$. Being in Class $\mathcal{R}$ and knowing n is not sufficient to perform a TDB in the two other modes (shortest and fastest). Being in Class $\mathcal{B}$ and knowing b is sufficient to make a *shortest TDB*, p in class $\mathcal{P}$ to be able to perform a *fastest TDB*. This therefore limits the feasibility of the *fastest TDB* to the class $\mathcal{P}$ and makes *shortest TDB* feasible in class $\mathcal{P}$ and in Class $\mathcal{B}$ in $O(nb)$. This led to show that the set of problems solvable in Class $\mathcal{R}$ knowing n is strictly included in the set of those solvable in Class $\mathcal{B}$ with a knowledge of b. This later set is strictly included in turn in the set of problems solvable in class $\mathcal{P}$ knowing p.

The distributed algorithms for TDB are described in Casteigts *et al.* [2014]. Besides the feasibility analysis, The authors are interested more generally in the message and time complexity of the algorithms. The more the algorithm has knowledge about the graph, the better we can do and the less will be the complexity. The algorithm uses two types of messages: information messages for the broadcast and control messages for termination detection. To reach all the graph nodes with a broadcast, $|E|$ information messages are needed in all cases. However, having more knowledge about the graph prevents some actions like exchange of some control messages. For example in the case of a *foremost TDB*, with the knowledge of b we go from $O(n^2)$ to $O(n)$ control messages. By knowing b and n, the termination is detected without an exchanging of control messages.

The authors describe the computational relationship between the three different problems by a hierarchical relation summarizing the results of the analysis. If we consider the universe of classes with different levels of knowledge $\mathcal{U} = \{\mathcal{R}_n, \mathcal{B}_n, \mathcal{B}_b, \mathcal{B}_{\{b,n\}}, \mathcal{P}_n, \mathcal{P}_b, \mathcal{P}_{\{b,n\}}, \mathcal{P}_p\}$ where C_k denotes class C with knowledge k, then *foremost TDB* $\prec$ *shortest TDB* $\prec$ *fastest TDB*, such that if $P \prec P'$ then P is feasible in C_k if P' is, and there exists $C'_{k'}$ in which P is feasible but not P'.

After the first broadcast, the computed tree can be reused by the succeeding broadcasts. In *shortest TDB* we are interested only in journeys length, this makes the resulting tree reusable for the subsequent broadcasts and reduces the amount of information messages from $O(m)$ to $O(n)$. However in *foremost* and fastest TDB computed trees can be reused only in class $\mathcal{P}$ with knowledge of p.

1.3 Automatic Classification

In addition to the usefulness of this dynamic graph hierarchy to transpose results and to compare problems, it provides a basis for an analysis process to verify the correct behavior of an algorithm in a given mobility context by testing the membership of this latter to the corresponding classes of dynamic graphs.

Casteigts *et al.* [2009] propose a framework that aims to characterize how appropriate a given algorithm is to a given dynamic context. The workflow is presented in Figure 1.6. A first step is that algorithms are analyzed to characterize necessary/sufficient conditions. This produces classes of dynamic graphs. Dynamic graphs instances can be obtained from network traces generated using mobility models and real-world networks. Then checking how given instances are spread on given classes can give an indication about the suitability of a given algorithm in a mobility context.

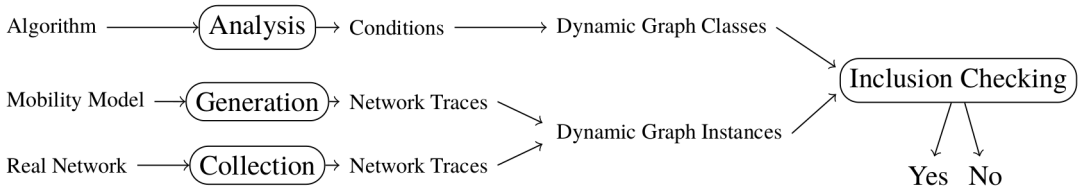


Figure 1.6: Automatic testing of algorithms relevance to dynamic contexts (Casteigts *et al.* [2009]).

A key issue is that of understanding how far such a framework could be automated. In this work we provide a contribution to the automation of the core operation of *Inclusion checking*. We will present, in this thesis, strategies for the automatic classification of dynamic graphs. We first focus on two specific classes (Chapters 2 and 3), then we provide a generic framework for testing properties in dynamic graphs (Chapter 4).

Chapter 5 discusses the case where no property is guaranteed (the graph is assumed to be in any class). Some concepts and notations that are used only in this chapter are defined there.

Chapter 2

Testing Temporal Connectivity

Contents

In this chapter, we address the problem of testing whether a given dynamic graph is temporally connected, *i.e.* a temporal path (also called a *journey*) exists between all pairs of nodes. We consider a discrete version of the problem, where the topology is given as an evolving graph $\mathcal{G} = \{G_1, G_2, ..., G_\delta\}$ see Chapter 1 for definitions). Two cases are studied, depending on whether a single edge or an unlimited number of edges can be crossed in a same G_i (strict journeys *vs* non-strict journeys).

The results presented in this chapter were published in AlgoTel (Barjon *et al.* [2014c]) and AETOS International Conference (RCFRUS) (Barjon *et al.* [2014b]).

2.1 Introduction

Connected and mobile devices such as mobile phones, satellites, cars, or robots form highly dynamic networks in which connectivity between nodes evolves rapidly and continuously. Furthermore, the topology of such a network at a given time is generally not connected, and even extremely sparse in the case of exploration or surveillance scenarios (Burgard *et al.* [2000]; Flocchini *et al.* [2013]), or when passive mobility is considered with humans or animals (Jea *et al.* [2005]; Shah *et al.* [2003]). However, even in these extreme cases, a form of connectivity arises over time and space, by means of delay tolerant communications, where messages are retained until an opportunity of transmission appears (mechanisms of type "*store-carry-forward*"). This type of connectivity is referred to as *temporal connectivity*, it is characterised by the fact that each node in a dynamic graph can join all other nodes by means of a temporal path. The concept of temporal connectivity is relatively old and dates back at least to the article Awerbuch and Even [1984b]. This property defines a class of dynamic graphs called temporally connected dynamic graphs (Class 4, Def. 1.16). In this chapter, the problem that we study is to test whether a given dynamic graph belongs to this class or not.

This chapter is organized as follows. In 2.2 we present the main definitions and makes some basic observations. In section 2.2.1 we present some existing works that can be adapted to solve the problem in the case of strict journeys. In Section 2.3 we propose a solution for strict temporal connectivity and non-strict temporal connectivity, and we compare its time complexity with that of the adapted solutions.

2.2 Model and Definitions

We consider, in this chapter, dynamic graphs that are given as untimed evolving graphs (Def. 1.4), that is, a sequence $\mathcal{G} = \{G_1, G_2, ..., G_\delta\}$ such that $G_i = (V, E_i)$ describes the network topology at (discrete) time i. We assume that the changes between two consecutive graphs are arbitrary. We distinguish two parameters to report on the number of edges in the graph: the maximal number of edges that exist at each single step, i.e. $\mu = max(|E_i|)$, and the total number of edges that exist over time, i.e. $m = |\cup E_i|$. Of course, whatever the considered graph, we have $m \geq \mu$. Moreover, as already discussed, it is common that the graph is sparse, it is not rare that a practical scenario verifies $\mu = o(n)$, or even $\mu = \Theta(1)$, while $m = \Theta(n \log n)$ or $m = \Theta(n^2)$.

We are interested in this chapter in the class of dynamic graphs *Temporal Connectivity* (Class 4, Def. 1.16):

Definition 2.1 (Temporal Connectivity, *Class 4*, Def. 1.16)**.** *A dynamic graph* $\mathcal{G} = \{G_i = (V, E_i)\}$ *is* temporally connected *iff for all pairs of nodes* $(u, v) \in V^2$*, there exist a journey* $u \rightsquigarrow v$*.*

Definition 2.2 (Strict Temporal Connectivity, *Class 5*, Def. 1.17)**.** *In the same way, a dynamic graph* $\mathcal{G} = \{G_i = (V, E_i)\}$ *is* strictly temporally connected *iff for all pairs of nodes* $(u, v) \in V^2$*, there exist a strict journey* $u \overset{st}{\rightsquigarrow} v$*.*

We consider, in this chapter, the problem of testing whether a dynamic graph is (strictly) temporally connected or not. In other words, we want to decide if there is a (strict) journey between every pair of nodes in a given dynamic graph.

Definition 2.3. *We will use* TEMPORAL-CONNECTIVITY *to refer to the problem of deciding if a dynamic graph* $\mathcal{G}$ *is temporally connected or not.*

Definition 2.4. *We will use* STRICT-TEMPORAL-CONNECTIVITY *to refer to the problem of deciding if a dynamic graph* $\mathcal{G}$ *is strictly temporally connected or not.*

A key concept for this problem is that of *transitive closure* of journeys, introduced in Bhadra and Ferreira [2003]. This is a static directed graph $\mathcal{G}^*$ whose edges represent the potential journeys in a dynamic graph $\mathcal{G}$. Once this structure is computed from the dynamic graph the property of temporal connectivity becomes trivial to test.

Definition 2.5 (Strict Transitive Closure)**.** *The* strict transitive closure *of journeys in* $\mathcal{G}$ *is the static directed graph* $\mathcal{G}^*_{st} = (V, E^*_{st})$ *where* $(u, v) \in E^*_{st} \Leftrightarrow u \overset{st}{\rightsquigarrow} v$*.*

Definition 2.6 (Non-strict Transitive Closure)**.** *We define, in the same way, the* non-strict transitive closure *of journeys in the dynamic graph* $\mathcal{G}$ *by the* static *directed graph* $\mathcal{G}^* = (V, E^*)$ *such that* $(u, v) \in E^* \Leftrightarrow u \rightsquigarrow v$*.*

It should be noted that the transitive closure graph of journeys in $\mathcal{G}$ is directed whatever the nature (directed or not) of the edges in $\mathcal{G}$. This is due to the temporal dimension that, by nature, implies an orientation.

We address in this chapter the computation of the transitive closure in the case of strict journeys ($\mathcal{G}^*_{st}$) or non-strict journeys ($\mathcal{G}^*$) given a dynamic graph $\mathcal{G}$. In the case of strict journey (resp. non-strict journey) a single edge (resp. several edges) of

G_i can be crossed. From the transitive closure structure, the membership of a given dynamic graph to several classes of dynamic graphs can be decided (Casteigts *et al.* [2009]).

- $\mathcal{G}^*$ contains an out-dominating set of size 1 $\Leftrightarrow \mathcal{G} \in$ *Class 1* $(\exists u \in V : \forall v \in V, u \rightsquigarrow v)$.
- $\mathcal{G}^*_{st}$ contains an out-dominating set of size 1 $\Leftrightarrow \mathcal{G} \in$ *Class 2* $(\exists u \in V : \forall v \in V, u \overset{st}{\rightsquigarrow} v)$.
- $\mathcal{G}^*$ contains an in-dominating set of size 1 $\Leftrightarrow \mathcal{G} \in$ *Class 3* $(\exists u \in V : \forall v \in V, v \rightsquigarrow u)$.

Observation 2.1. *Regarding our problem, testing whether a dynamic graph is (strictly) temporally connected comes to test whether its (strict) transitive closure is complete.*

- $\mathcal{G}^*$ *is a complete graph* $\Leftrightarrow \mathcal{G} \in$ *Class 4* $(\forall u, v \in V, u \rightsquigarrow v)$.
- $\mathcal{G}^*_{st}$ *is a complete graph* $\Leftrightarrow \mathcal{G} \in$ *Class 5* $(\forall u, v \in V, u \overset{st}{\rightsquigarrow} v)$.

Figure 2.1 shows the strict and non-strict transitive closures of journeys in the dynamic graph $\mathcal{G}$ presented in the same Figure. Each edge on $\mathcal{G}^*$ (resp. $\mathcal{G}^*_{st}$) represents the existence of a journey (resp. strict journey) in $\mathcal{G}$. In $\mathcal{G}^*_{st}$, the absence of $(e, a), (e, b)$ and (d, e) indicates the non-existence of strict journeys from e to a and b and from b to e in $\mathcal{G}$ ($e \overset{st}{\rightsquigarrow} a, e \overset{st}{\rightsquigarrow} b$ and $d \overset{st}{\rightsquigarrow} e$). As the temporal connectivity of a dynamic graph is equivalent to the completeness of its transitive closure, the absence of at least one edge in $\mathcal{G}^*_{st}$ asserts that the dynamic graph $\mathcal{G}$ is not strictly temporally connected. By definition, a strict journey is a non-strict journey. This implies that a strict transitive closure of journeys in a dynamic graph $\mathcal{G}$ is always contained in its non-strict transitive closure. In our example $\mathcal{G}^*$ contains, in addition to the strict transitive closure $\mathcal{G}^*_{st}$, all the edges corresponding to the impossible strict journeys in $\mathcal{G}$: $(e, a), (e, b)$ and (d, e) as a result of the presence of the respective non-strict journeys $e \rightsquigarrow a$, $a \rightsquigarrow b$ and $b \rightsquigarrow e$. This makes $\mathcal{G}^*$ complete. Therefore, $\mathcal{G}$ is temporally connected but not strictly temporally connected.

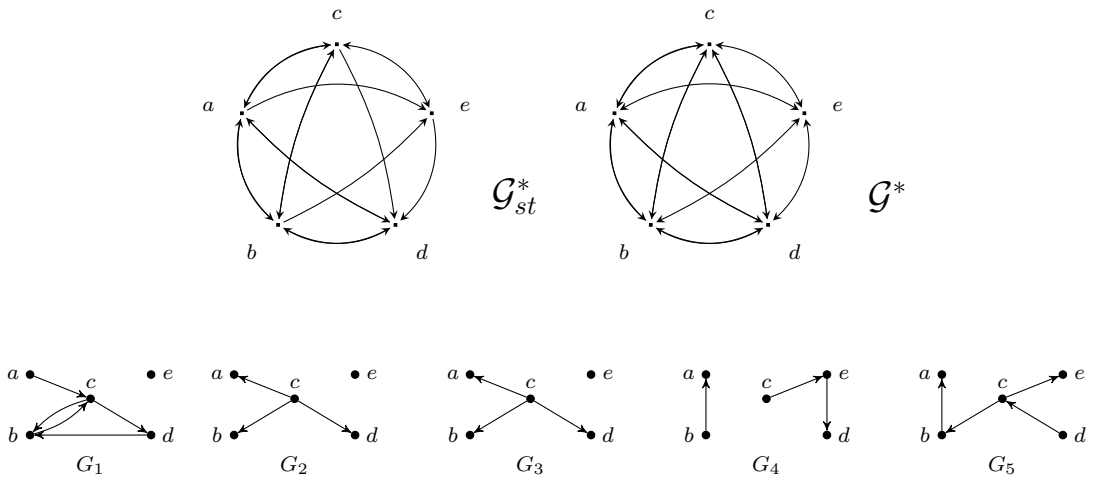


Figure 2.1: Example of strict ($\mathcal{G}^*_{st}$) and non-strict ($\mathcal{G}^*$) transitive closure of journeys in a dynamic graph $\mathcal{G}$.

2.2.1 Related Works

Different studies have addressed problems related to testing connectivity or related problems in dynamic graphs in literature. In the case of strict journeys, several algorithms can be adapted to compute $\mathcal{G}^*_{st}$.

1. Different algorithms are given in Bui-Xuan *et al.* [2003], each computing optimal journeys according to a given criterion (foremost, shortest, fastest) in a timed evolving graph model. Any of these algorithms can be adapted to compute $\mathcal{G}^*_{st}$ by using the appropriate parameters. Precisely, these algorithms compute the journeys taking into account the duration of edges crossing (latency) and the duration of each graph G_i. If we assign to each G_i a unit duration that also corresponds to the duration of edges crossing, then, for a given source node, the result is the set of strict journeys. The algorithm has to be executed n times, once from each node, in order to compute the transitive closure of the journeys. The most efficient of the three algorithms (foremost journeys) has an execution time of $O(m \log \delta + n \log n)$, hence a total time of $O(n(m \log \delta + n \log n))$.

2. An algorithm computing a generalization of the transitive closure of journeys was proposed in Whitbeck *et al.* [2012]. This generalization, called *dynamic reachability graph*, corresponds to a transitive closure of journeys parametrized by a starting date, a maximal duration of the journeys, and a traversal time for edges. It applies to dynamic graphs represented as TVGs

($\mathbb{T} = \mathbb{R}^+$ in the case of Whitbeck *et al.* [2012]). This algorithm can be used to compute $\mathcal{G}^*_{st}$ as follows:

- First create a TVG whose edges presence dates (function ρ) are all multiples of some unit value that also corresponds to the latency given by function ζ (here, a constant);
- Then, the constraint on journeys duration is set to $+\infty$ and the departure date is set to 0;
- Finally $\mathcal{G}^*_{st}$ is obtained by executing the algorithm from Whitbeck *et al.* [2012] on the created TVG.

Informally, the strategy of that algorithm is to compose reachability graphs incrementally over increasing periods of time, namely each graph covering 2^i time steps is obtained by composition of two graphs covering 2^{i-1} time steps (for i from 1 to $\log \delta$). The complexity of this algorithm is $O(\delta \log \delta\, mn \log n)$. The authors do not exclude the possibility to get rid of the trailing $\log n$ factor, potentially linked to an implementation choice (see Section 4.3 of Whitbeck *et al.* [2012]).

3. Various adaptations of the Bellman-Ford principle to dynamic graphs, such as Kossinets *et al.* [2008] in the case of link stream. One component of our algorithm also falls into this category.

2.3 Testing Temporal Connectivity in Sparse Dynamic Graphs

We propose, to test whether a graph is (strictly) temporally connected, a dedicated approach for computing the transitive closure (strict at first) of journeys in an untimed directed evolving graph $\mathcal{G}$ which has a better time complexity than the adaptation of Whitbeck *et al.* [2012], and than the adaptation of Bui-Xuan *et al.* [2003] for a range of dynamic graphs, in particular those whose density is low at any time (sparse dynamic graphs), though arbitrarily dense over time. The algorithm consists of a temporal adaptation of the Bellman-Ford principle used in static graphs to compute distances between nodes. This principle was also adapted in Kossinets *et al.* [2008] to compute time lags between entities based on a contact history (e.g. a sequence of dated emails). Our adaptation is quite straight and its time complexity is $O(\delta \mu n)$ with the considered data structure (a mere sequence of sets of edges). As discussed above, the distinction between μ and m is relevant in a number of scenarios based on mobile communicating entities. Furthermore, this type

of graphs typically corresponds to those in which the question of the temporal connectivity occurs, since it is not *a priori* granted. In the case of non-strict journeys, for which we do not know any existing algorithm, we show that the graph can be pre-processed in such a way that the same solution can be directly adapted with the same time complexity: $O(\delta\mu n)$. This variant is based on a double transitive closure: a *static* transitive closure applied to each G_i independently, and a temporal one (as in the case of strict journeys) applied to the sequence of those static closures.

Both algorithms gradually build the transitive closure as the edges are examined; these are *online* algorithms. They stop their execution as soon as the temporal connectivity is satisfied (or after the whole dynamic graph has been examined). A by-product of the execution is to make $\mathcal{G}^*_{st}$ and $\mathcal{G}^*$ available for further connectivity queries, these queries being then reduced to simple adjacency tests in a static graph. Time complexity is analysed throughout the chapter. We provide in Section 2.3.3 a more detailed comparison that indicates the values of δ, μ and m for which our solution performs better than the adaptation from Bui-Xuan *et al.* [2003].

2.3.1 Computation of the Transitive Closure for Strict Journeys *(strict transitive closure)*

We propose below an algorithm for computing the strict transitive closure $\mathcal{G}^*_{st}$ in the general case where $\mathcal{G}$ is directed. The principle of the algorithm is to build, step by step, the set of all the predecessors of each node v, i.e., the set $\{u : u \overset{st}{\rightsquigarrow} v\}$. Each step i of the algorithm works on a static graph G_i of $\mathcal{G}$. Let $\mathcal{P}(v,t)$ be the set of known predecessors of v by the end of the t first steps of the algorithm (i.e. after taking into account edge sets: $E_1, ..., E_t$). The core of step i is to add $\mathcal{P}(u, i-1)$ to $\mathcal{P}(v,i)$ for each edge $(u,v) \in E_i$. In practice, only two variables $\mathcal{P}(v)$ and $\mathcal{P}^+(v)$ are maintained for each node v. $\mathcal{P}^+(v)$ contains the new predecessors of v (computed during the current step). At the end of the current step $\mathcal{P}^+(v)$ is merged to $\mathcal{P}(v)$, the set of all predecessors of v. The detailed operations are given in Algorithm 1.

We show in Figure 2.2 an example of a strict transitive closure computation. In the example, the graph $\mathcal{G} = \{G_i = (V, E_i)\}$ of length $\delta = 4$ is represented in the first line of the table. The second line of the table shows the four computation steps results. At each step i (for each graph G_i), the set of predecessors of each node is computed according to the algorithm: $\mathcal{P}_i(v) = \mathcal{P}_{i-1}(v) \cup \mathcal{P}_{i-1}(u)$ for all $(u,v) \in E_i$. We also show the evolution of a's predecessors set $\mathcal{P}_4(a)$ computation by a bold text that points the involved sets. $\mathcal{P}_4(a)$ is the final output used to compute the strict-transitive closure $\mathcal{G}^*_{st}$ and it corresponds to the set of a's predecessors up to G_4. This output is a result of a series of computations. Let's start from this final result and go back to the initial values to illustrate the computation process. In G_4, the only incoming edge on a is (b,a) represented with a thick edge, $\mathcal{P}_4(a)$

in this case (4^{th} step) is computed from $\mathcal{P}_3(b)$ which is merged with $\mathcal{P}_3(a)$. At this step $\mathcal{P}_3(a)$ contains already $\mathcal{P}_3(b)$, this can be explained by the the fact that $\mathcal{P}(b)$ remains unchanged in the steps 2 and 3 ($\mathcal{P}_3(b) = \mathcal{P}_2(b)$) due to the absence of an incoming edge on b in $\mathcal{G}_3$. So, the value of $\mathcal{P}_3(a)$ is retained for $\mathcal{P}_4(a)$. The same operation is applied at the 3^{rd} step since the state of a in the graph G_3 is the same as in G_4, which adds the node d to $\mathcal{P}_2(a)$ to give as a result $\mathcal{P}_3(a)$. $\mathcal{P}_2(b)$ is computed from $\mathcal{P}_1(c)$ merged with $\mathcal{P}_1(b)$ upon the appearance of the edge (c, b) in G_2. In the same step, $\mathcal{P}_2(a)$ is computed in the same way because of the presence of (c, a) in G_2 but this time by merging $\mathcal{P}_1(a)$ with $\mathcal{P}_1(c)$. Following the same principle, $\mathcal{P}_1(a)$, $\mathcal{P}_1(b)$ and $\mathcal{P}_1(c)$ are computed from the initial sets $\mathcal{P}_0(a)$, $\mathcal{P}_0(b)$, $\mathcal{P}_0(c)$ and $\mathcal{P}_0(d)$. Similarly, the last sets of predecessors: $\{\mathcal{P}_4(v) : v \in V\}$ of all nodes are computed (using the same steps), then it is used to get the strict transitive closure $\mathcal{G}^*_{st} = (V, E^*_{st})$ of $\mathcal{G}$ where $(u, v) \in E^*_{st} \Leftrightarrow u \in \mathcal{P}_4(v)$. It can be seen in our case on the transitive closure, from the fact that it is not a complete graph, that the dynamic graph $\mathcal{G}$ is not strictly temporally connected.

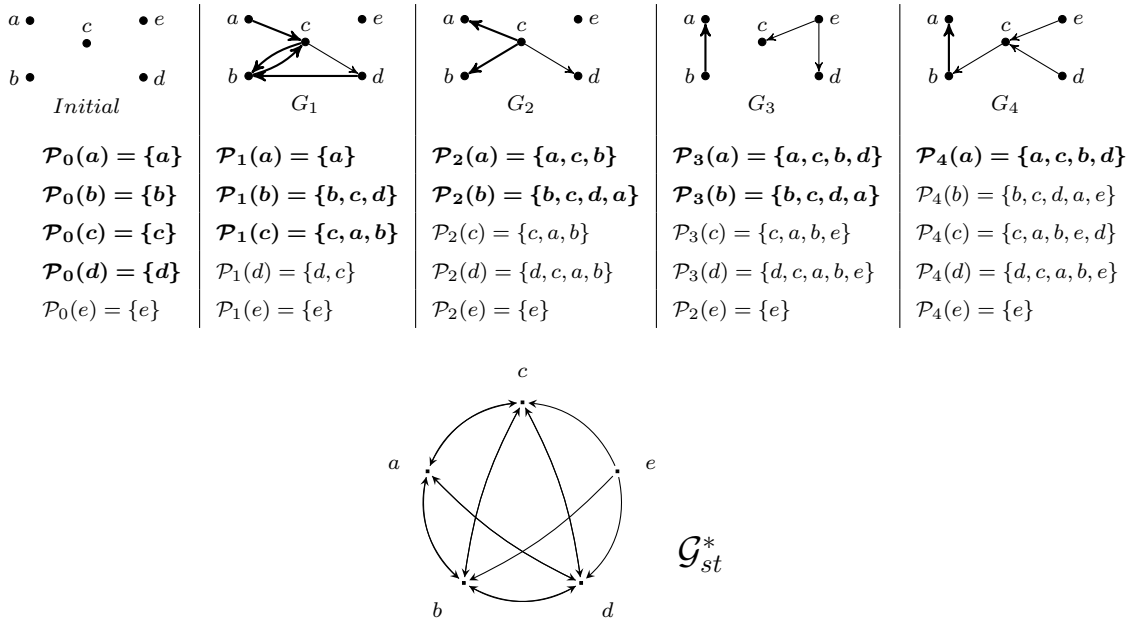


Figure 2.2: Example of strict transitive closure computation.

Time complexity

This section provides an analysis of the time complexity of the algorithm. In this analysis, we consider the use of *set* data structures, for which the union has at worst a linear cost in the number of items of the two considered sets. We also assume that the size of a set can be known in constant time, which is the case with most of the existing libraries that implements this kind of data structure (this value being

Input : A dynamic graph $\mathcal{G}$ given as $(V, \{E_i\})$
Output: A set of edges E^* such that $\mathcal{G}^*_{st} = (V, E^*)$

```
   // Initialization
1  foreach v in V do
2  |  P(v) ← {v};        // Each node is its own predecessor
3  |  P+(v) ← ∅;
4  foreach E_i in {E_i} do
5  |  UpdateV ← ∅ ;     // List of nodes whose predecessors
   |  will be updated
   |  // List predecessors induced by the edges in E_i
6  |  foreach (u, v) in E_i do
7  |  |  P+(v) ← P+(v) ∪ P(u);
8  |  |  UpdateV ← UpdateV ∪ {v}
   |  // Add found predecessors to known predecessors
9  |  foreach v in UpdateV do
10 |  |  P(v) ← P(v) ∪ P+(v);
11 |  |  P+(v) ← ∅;
   |  // Test whether transitive closure is complete;
   |     if so, terminates
12 |  isComplete ← true;
13 |  foreach v in V do
14 |  |  if |P(v)| < |V| then
15 |  |  |  isComplete ← false;
16 |  |  |  break;
17 |  if isComplete then
   |  |  // The algorithm terminates returning a
   |  |     complete graph (edges)
18 |  |  return V × V \ {loops}
   // Build transitive closure based on predecessors
19 E* ← ∅
20 foreach v in V do
21 |  foreach u in P(v) \ {v} do
22 |  |  E* ← E* ∪ (u, v)
23 return E*
```

Algorithm 1: Computation of the strict transitive closure $\mathcal{G}^*_{st}$

maintained as the set is modified).

Lemma 2.1. *For all $v \in V$, $|\mathcal{P}(v)| \leq \delta\mu$, i.e., a node cannot have more than $\delta\mu$ predecessors.*

Proof (by contradiction). If there is a node v such that $|\mathcal{P}(v) \setminus v| > \delta\mu$, then, by definition, there exists more than $\delta\mu$ nodes u different from v such that $u \rightsquigarrow v$. Each of these nodes is thus the origin of at least one edge, which means that more than $\delta\mu$ distinct edges existed. □

Theorem 2.1. STRICT-TEMPORAL-CONNECTIVITY *can be solved in* $O(\delta\mu n)$ *using Algorithm 1 that computes the strict transitive closure of journeys in a graph* $\mathcal{G}$.

Proof. The initialization loop (line 1) is linear in n. The main loop (line 4) iterates as many times as the number of steps in $\mathcal{G}$, i.e. δ times. The main loop has three sub-loops (lines 6, 9 and 13), each being dominated by $O(|E_i| \cdot n) = O(\mu n)$. Finally, the construction of the transitive closure (line 20) , if it is not complete before the end (worst case), consists of a loop that, for each node, iterates over its predecessors. Since the number of predecessor of a given node cannot exceed $\delta\mu$ (Lemma 2.1), this latter loop is also dominated by $O(\delta\mu n)$. □

2.3.2 Computation of the Transitive Closure for Non-strict Journeys (non-strict transitive closure)

In this section, we focus on the computation of $\mathcal{G}^*$, i.e. the transitive closure of the journeys for which an unlimited number of edges can be crossed at each step (non-strict journey). Indeed, the relaxation of the constraint that the journeys are strict implies that at each step i, if a path (in the classic acceptance of the word) exists from u to v, then u can join v at the same step. A simple observation allows us to reuse Algorithm 1 almost directly. The algorithm therefore consists in pre-computing, at each step, the transitive closure, in the classic static meaning of this term, of the edges present in G_i, resulting in a graph G_i^*, each edge of which corresponds to a path in G_i. Then Algorithm 1, applied to the dynamic graph $\{G_i^*\}$, produces directly the non-strict transitive closure $\mathcal{G}^*$. In Figure 2.3, $\{G_i^*\}$ (second line of the table) represents the sequence of static paths transitive closures computed from the dynamic graph $\mathcal{G}$ (first line of the table). $\mathcal{G}^*$ is the result of applying our algorithm for strict transitive closure (Algorithm 1) on $\{G_i^*\}$. In some way, the non-strict transitive closure $\mathcal{G}^*$ of journeys in a dynamic graph $\mathcal{G}$ is the strict transitive closure of journeys in its sequence of static transitive closures $\{G_i^*\}$.

Time complexity

Theorem 2.2. TEMPORAL-CONNECTIVITY *can be solved in* $O(\delta\mu n)$.

As we use the same algorithm for strict transitive closure on the sequence of static transitive closure $\{G_i^*\}$, the time complexity of this algorithm essentially depends on the cost of the computation of the static transitive closure G_i^* of journeys

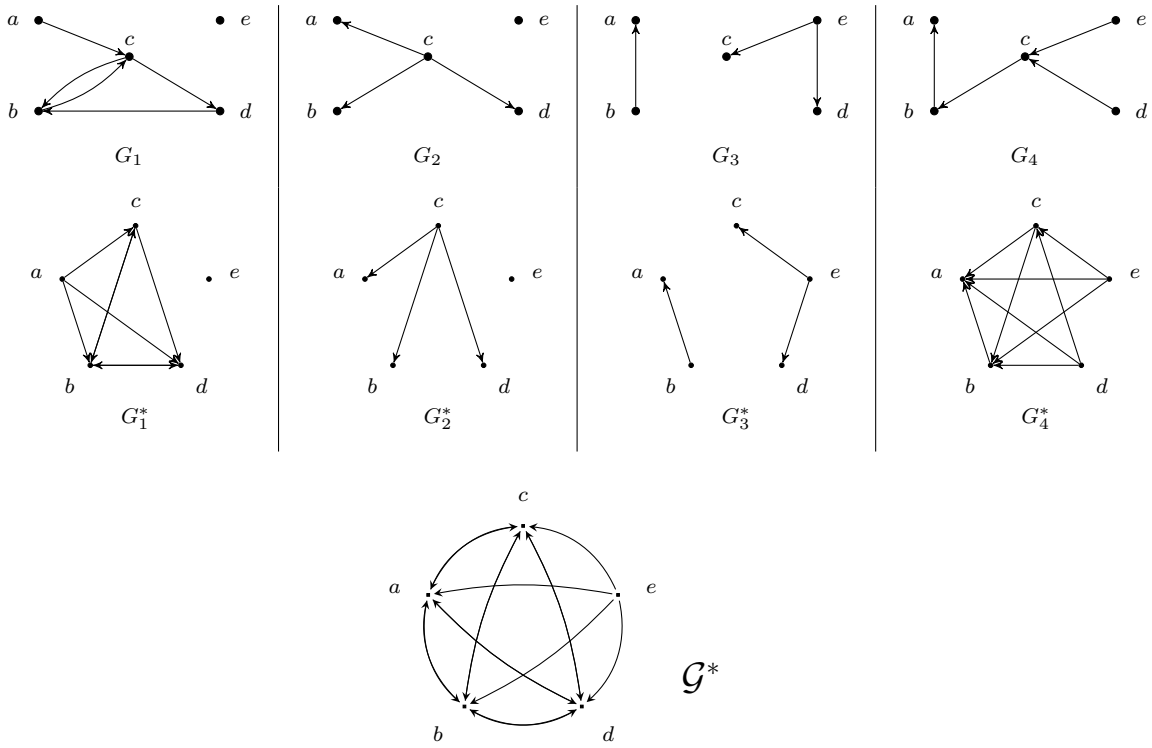


Figure 2.3: Example of non-strict transitive closure computation.

in the graphs G_i. This can be done by a depth first search (DFS) or by a breadth first search (BFS) run from each node in G_i. Each of these runs having an execution cost in $O(|E_i|) = O(\mu)$ and thus, the extra cost of this operation remains within $O(\delta\mu n)$ time.

2.3.3 Comparison

This section compares the complexity of the proposed algorithm with the adaptation of that from Bui-Xuan *et al.* [2003] (based on foremost journeys), which has a running time of $O(n(m \log \delta + n \log n))$, where $m \neq \mu$ is the total number of edges existing over time, i.e. $|\cup E_i|$.

The question is therefore to compare this complexity to $O(\delta\mu n)$, or after simplification by n, to compare $O(\delta\mu)$ to $O(m \log \delta + n \log n)$. These complexities belong to a four-dimensional space : μ, m, δ and n; it is therefore not easy to compare them. We propose to study them asymptotically in n, by varying the values of μ, m and δ. Precisely, we vary the order of μ and m (instant density *vs.* cumulated density) for several ratios of possibles values of δ and n (i.e. the length of the sequence $\mathcal{G}$ in function of n). Table 2.1 contains 60 results, including a dozen that show the

transition in efficiency between both solutions (the others can be extrapolated without computation by considering the relative impact of factors δ and $\log \delta$ in both formulas). The symbols $-$, $+$ and $\approx$ indicate respectively the ranges of values for which our solution has a lower, higher asymptotic complexity and a complexity of the same order. Empty cells at the right of a + (resp. left of a n/a) are filled with + (resp. n/a). To make the verification of these results simpler, we provide in the right column an intermediate expression, obtained after replacing μ and m in both expressions $O(\delta\mu)$ and $O(m \log \delta + n \log n)$.

In summary, the table confirms that the proposed solution becomes more relevant as the difference between instant density and cumulated density increases, which is not surprising. It is also not surprising, given the presence of the factor δ versus $\log \delta$, that our solution is less efficient when the number of time steps increases. The table reveals some ranges of realistic values where the proposed solution behaves better than the other, for instance when the values of μ, m, and δ are respectively $(O(n), \Theta(n^2), O(n))$; or $(O(\log n), \Omega(n \log n), O(n))$; or $(O(\sqrt{n})), \Omega(n), O(\sqrt{n}))$. Finally, the fact that the algorithm terminates as soon as temporal connectivity is satisfied allows us to put in perspective the impact of parameter δ.

$\mu = \Theta(.)$	$m = \Theta(.)$	$\delta = \Theta(\log n)$	$\delta = \Theta(\sqrt{n})$	$\delta = \Theta(n)$	$\delta = \Theta(n^2)$	Intermediate computation $\Theta(.) \pm \Theta(.)$
$\log n$	n		n/a	$\approx$	$+$	$\delta \log n \pm n \log \delta + n \log n$
$\sqrt{n}$	n	n/a	$-$	$+$		$\delta\sqrt{n} \pm n \log \delta + n \log n$
n	n	$\approx$	$+$			$\delta n \pm n \log \delta + n \log n$
$\log n$	$n \log n$		n/a	$-$	$+$	$\delta \log n \pm (n \log n) \log \delta$
$\sqrt{n}$	$n \log n$		n/a	$+$		$\delta\sqrt{n} \pm (n \log n) \log \delta$
n	$n \log n$	$-$	$+$	$+$		$\delta n \pm (n \log n) \log \delta$
$n \log n$	$n \log n$	$+$				$\delta \pm \log \delta$
$\log n$	n^2			n/a	$\approx$	$\delta \log n \pm n^2 \log \delta$
$\sqrt{n}$	n^2			n/a	$+$	$\delta\sqrt{n} \pm n^2 \log \delta$
n	n^2		n/a	$-$	$+$	$\delta n \pm n^2 \log \delta$
$n \log n$	n^2		n/a	$\approx$	$+$	$\delta(n \log n) \pm n^2 \log \delta$
$n\sqrt{n}$	n^2	n/a	$-$	$+$		$\delta(n\sqrt{n}) \pm n^2 \log \delta$
n^2	n^2	$+$				$\delta n^2 \pm n^2 \log \delta$

Table 2.1: Running time comparison between the proposed algorithm and the adaptation of the algorithm from Bui-Xuan *et al.* [2003].

Conclusion

In this chapter we addressed the problem of testing whether a given dynamic graph is temporally connected, *i.e.* a temporal path (also called a *journey*) exists between all pairs of nodes. We considered a discrete version of the problem, where the topology is given as an evolving graph $\mathcal{G} = \{G_1, G_2, ..., G_\delta\}$ whose set of nodes is invariant and the set of (directed) edges varies over time. Two cases have been studied, depending on whether a single edge or an unlimited number of edges can

be crossed in a same G_i (strict journeys *vs* non-strict journeys).

In the case of *strict* journeys, we investigated algorithmic solutions based on the computation of the transitive closure of journeys in the graph that have time complexity of $O(\delta\mu n)$. In this case of (*strict* journeys), a number of existing algorithms designed for more general problems can be adapted (Bui-Xuan *et al.* [2003]). We adapted one of them to the above formulation of the problem and characterized its running time complexity. The parameters of interest were the length of the graph sequence $\delta = |\mathcal{G}|$, the maximum *instant* density $\mu = max(|E_i|)$, and the *cumulated* density $m = |\cup E_i|$. We proved that our algorithm has a time complexity of $O(\delta\mu n)$, where n is the number of nodes. We compared the complexity of our algorithm to that of the adaptation of the other solutions: the solution presented in Whitbeck *et al.* [2012] is always more costly (keep in mind that it solves a more general problem), the other solution presented in Bui-Xuan *et al.* [2003] is more or less costly depending on the interplay between instant density and cumulated density. The length δ of the sequence also plays a role. We characterized the key values of δ, μ and m for which either algorithm should be used. Our solution is shown to be relevant for sparse mobility scenario (e.g. robots or UAVs exploring an area) where the number of neighbors at a given time is low, though many nodes can be seen over the whole execution.

In the case of *non-strict* journeys, for which no algorithm is known, we showed that some pre-processing of the input graph allows us to re-use the same algorithm than before. By chance, these operations happens to cost again $O(\delta\mu n)$ time, which implies that the second problem is not more difficult than the first.

Chapter 3

Testing T-interval Connectivity

Contents

In this chapter, we look at the problem of deciding whether a given sequence $\mathcal{G}$ is T-interval connected for a given T. We also consider the related problem of finding the largest T for which the given $\mathcal{G}$ is T-interval connected. We focus on high-level strategies that consider graphs as block boxes. We first show that both problems require $\Omega(\delta)$ elementary operations using the basic argument that every graph of the sequence must be considered at least once. More surprisingly, we show that both problems can be solved using only $O(\delta)$ such operations and we develop optimal online algorithms that achieve these matching bounds. Hence, the cost of the operations – both of them linear in the number of edges – is counterbalanced by efficient high-level logic that could, for instance, benefit from dedicated circuits (or optimized code) for both operations.

The chapter is organized as follows. Section 3.1 presents the main definitions and makes some basic observations, including the fact that both problems can be solved using $O(\delta^2)$ elementary operations by a naive strategy that examines $O(\delta^2)$ intermediate graphs. Section 3.2 presents a second strategy, yielding upper bounds of $O(\delta \log \delta)$ operations for both problems. Its main interest is in the fact that it can be parallelized, and this allows us to classify both problems as being in **NC** (i.e. Nick's class). In Section 3.3 we present an optimal strategy which we use to solve both problems online in $O(\delta)$ operations. This strategy exploits structural properties of the problems to construct carefully selected subsequences of the intermediate graphs. In particular, only $O(\delta)$ of the $O(\delta^2)$ intermediate graphs are selected for evaluation by the algorithms. In Section 3.4, we extend our online algorithms to a dynamic setting in which the measure of connectivity is based on the recent evolution of the network.

The results presented in this chapter were published in the 9th International Conference on Algorithms and Complexity CIAC (Casteigts *et al.* [2015a]) and AlgoTel (Casteigts *et al.* [2015b]).

3.1 Model, Definitions and Basic Observations

In this chapter, we consider dynamic graphs that are given as untimed evolving graphs (Def. 1.4), that is, a sequence $\mathcal{G} = \{G_1, G_2, ..., G_\delta\}$ of length δ such that $G_i = (V, E_i)$ describes the network topology at (discrete) time i. We assume that the changes between two consecutive graphs are arbitrary.

In a dynamic context, an important analysis would be to measure a form of stability of the graph. We suppose that it is too strong and meaningless to consider the overall stability of the entire graph for a period, it would remove even the dynamic aspect from the graph. However, since one of the most important properties of a dynamic graph is the connectivity, it would be interesting to consider a form of stability based on the existence of a spanning connected component for a duration.

The class of T-*interval connected* graphs was identified in Kuhn *et al.* [2010] as playing an important role in several distributed problems, such as determining the size of a network or computing a function of the initial inputs of the nodes. Informally, T-interval connectivity requires that, for every T consecutive graphs in the sequence $\mathcal{G}$, there exists a common connected spanning subgraph.

Kuhn *et al.* [2010] focus on two problems: the counting problem which consists in computing the size of the graph and the k-token dissemination problem in which k pieces of information, initially on k nodes, must be collected by all the

nodes of the graph. For for k-token dissemination, the authors were interested in the case where $k = n$ with, initially, one piece of information on each node. To solve this problem, nodes need to determine the size of the graph in order to know the number of pieces of information to collect. The resolution of this problem allows the computation of any function of the initial states of the nodes. It was shown that both problems can be solved in $O(n^2)$ rounds in 1-interval connected graphs. For T-interval connected graphs with $T > 1$, they show that both problems can be solved in $O(min\{n^2, n + n^2 \log(n)/T\})$ rounds with unknown T. If T is known, they prove that both problems are solvable in $O(n+n^2/T)$ time complexity. Results are presented on a model where nodes can start the computation at different rounds if the graph is 2-interval connected.

The class of T-interval connected dynamic graphs generalizes the class of dynamic graphs that are connected at all time instants (O'Dell and Wattenhofer [2005]). Indeed, the latter corresponds to the case that $T = 1$. From a set-theoretic viewpoint, however, every $T > 1$ induces a class of graphs that is a strict subset of the class in O'Dell and Wattenhofer [2005] because a graph that is T-interval connected is obviously 1-interval connected.

Let us start by defining T-interval connectivity formally:

Definition 3.1 (***T***-interval connectivity , def. 1.24)**.** *A dynamic graph $\mathcal{G}$ is said to be* T-interval connected *if for every $t \in [1, \delta - T + 1]$, all graphs in $\{G_t, G_{t+1}, ..., G_{t+T-1}\}$ share a common connected spanning subgraph. In other words, every sequence of length T in the dynamic graph $\mathcal{G}$ shares a common connected spanning subgraph.*

Throughout the chapter we consider *undirected* edges, which is the setting in which T-interval connectivity was originally introduced. We do not make any assumptions about the data structures that are used to represent the sequence of graphs. As such, we focus on high-level strategies that work directly at the graph level. The fact that our algorithms are high-level allows them to work exactly the same for T-interval strong connectivity, which is the analogue of T-interval connectivity for directed graphs (Kuhn *et al.* [2010]) (i.e. for every $t \in [1, \delta - T + 1]$, all graphs in $\{G_t, G_{t+1}, ..., G_{t+T-1}\}$ share a common connected strongly spanning subgraph), provided that *connectivity test* operation is replaced by *strong connectivity test*. As we shall discuss, these operations have linear cost in the number of edges in both directed and undirected graphs.

Figure 3.1 shows an Example of a 4-interval connected dynamic graph $\mathcal{G}$ of length $\delta = 8$. In $\mathcal{G}$ each sequence $(G_i, G_{i+1}, G_{i+2}, G_{i+3})$ of length 4 has a common spanning subgraph $G_{(i,i+3)}$ for $1 \leq i \leq 5$. In the general case $G_{(i,j)}$ is called *intersection graph* and it is defined as follows:

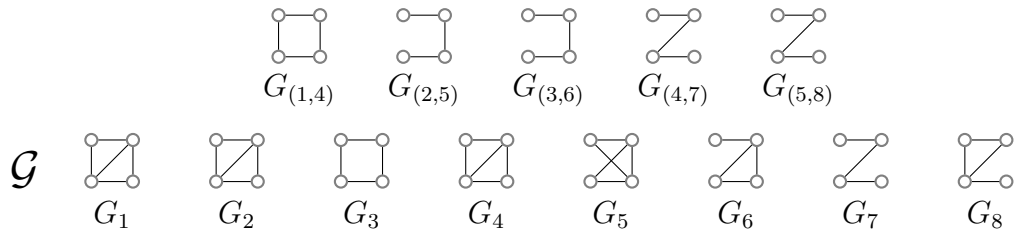


Figure 3.1: Example of a 4-interval connected dynamic graph $\mathcal{G}$ of length $\delta = 8$.

Definition 3.2 (Intersection graph)**.** *Given a (finite) set S of graphs $\{G_i = (V, E_i), G_j = (V, E_j), \ldots, G_k = (V, E_k)\}$, we call the graph $(V, \cap\{E_i, E_j, \ldots, E_k\})$ the* intersection graph *of S and denote it by $\cap\{G_i, G_j, \ldots G_k\}$. When the set consists of only two graphs, we talk about* binary intersection *and use the infix notation $G_i \cap G_j$. If the intersection involves a consecutive subsequence $\{G_i, G_{i+1}, \ldots, G_j\}$ of a dynamic graph $\mathcal{G}$, then we denote the intersection graph $\cap\{G_i, G_{i+1}, \ldots, G_j\}$ simply as $G_{(i,j)}$ and we say that $G_{(i,j)}$* covers *the sequence $\{G_i, G_{i+1}, \ldots, G_j\}$. We say that the intersection graphs $\{G_{(i,j)}, G_{(i',j')}, \ldots, G_{(i'',j'')}\}$* cover *the sequence $\{G_i, G_{i+1}, \ldots, G_{j''}\}$ if $\cap\{G_{(i,j)}, G_{(i',j')}, \ldots, G_{(i'',j'')}\} = \cap\{G_i, G_{i+1}, \ldots, G_{j''}\}$. We define the* size *of an intersection graph as the length of the sequence that it covers, the size of $G_{(i,j)}$ is $j - i + 1$.*

We consider in this chapter the decision problem of testing whether a dynamic graph $\mathcal{G}$ is T-interval connected or not. We are also interested in the related problem of finding the largest T for which $\mathcal{G}$ is T-interval connected.

Definition 3.3 (Testing $\boldsymbol{T}$-interval connectivity)**.** *We denote as T-*INTERVAL-CONNECTIVITY *the problem of deciding whether a dynamic graph $\mathcal{G}$ is T-interval connected for a given T.*

Definition 3.4 (Testing Interval connectivity)**.** *We denote as* INTERVAL-CONNECTIVITY *the problem of finding $\max\{T : \mathcal{G}$ is T-interval connected$\}$ for a given $\mathcal{G}$.*

Definition 3.5 (Intersection hierarchy)**.** *We call an intersection hierarchy the structure formed by the superposition of rows represented by sequences of intersection graphs of the same size. Let $\mathcal{G}^T = \{G_{(1,T)}, G_{(2,T+1)}, ..., G_{(\delta-T+1,\delta)}\}$. We call $\mathcal{G}^T$ the T^{th}* row *in $\mathcal{G}$'s intersection hierarchy, as depicted in Fig. 3.2. A particular case is $\mathcal{G}^1 = \mathcal{G}$. For any $1 \leq i \leq \delta - T + 1$, we define $\mathcal{G}^T[i] = G_{(i,i+T-1)}$. We call $\mathcal{G}^T[i]$ the i^{th}* element of row $\mathcal{G}^T$ *and i is called the* index of $\mathcal{G}^T[i]$ in row $\mathcal{G}^T$.

Figure 3.2 shows an example of an intersection hierarchy for a given dynamic graph $\mathcal{G}$ of length $\delta = 8$. In the example, $\mathcal{G}$ is 3-interval connected, but not 4-interval connected; $\mathcal{G}^4$ contains a disconnected graph $G_{(2,5)}$ because G_2, G_3, G_4, G_5 share no connected spanning subgraph.

Observation 3.1 (T-interval connectivity in the intersection hierarchy)**.** *A dynamic graph $\mathcal{G}$ is T-interval connected if and only if all graphs in $\mathcal{G}^T$ are connected. Formally, a dynamic graph $\mathcal{G}$ is T-interval connected if and only if the intersection graph $G_{(t,t+T-1)}$ is connected for every $t \in [1, \delta - T + 1]$.*

3.1.1 Bound on Computation Time

As shown in Observation 3.1, the concept of T-interval connectivity can be reformulated quite naturally in terms of the connectivity of some intersection graphs. For this reason, we consider two building block operations: *binary intersection* (given two graphs, compute their intersection) and *connectivity testing* (given a graph, test if it is connected). This approach is suitable for a high-level study of these problems when the details of changes between successive graphs in a sequence are arbitrary. If more structural information about the evolution of the dynamic graphs is known, for example, if it is known that the number of changes between each pair of consecutive graphs is bounded by a constant, then other algorithms could benefit from the use of sophisticated data structures and a lower-level approach might be more appropriate.

Cost of the operations: Using an adjacency list data structure for the graphs, a binary intersection can be performed in linear time in the number of edges.

Checking connectivity of a graph can also be done in linear time in the number of edges. In the case of undirected graphs, it can be done by building a depth-first search tree from an arbitrary root node and testing whether all nodes are reachable from the root node. Tarjan's algorithm for strongly connected components can be used for directed graphs.

Hence, both the *intersection* operation and the *connectivity testing* operation have similar costs. In what follows, we will refer to them as *elementary* operations.

One advantage of using these elementary operations is that the high-level logic of the algorithms becomes elegant and simple. Also, their cost can be counterbalanced by the fact that they are highly generic and thus could benefit from dedicated circuits (e.g., FPGA) or optimized code.

Naive Upper Bound: It suffices to compute the rows of $\mathcal{G}'s$ intersection hierarchy incrementally using the fact that each graph $G_{(i,i+k)}$ can be obtained as $G_{(i,i+k-1)} \cap G_{(i+1,i+k)}$. For instance, $G_{(3,6)} = G_{(3,5)} \cap G_{(4,6)}$ in Fig. 3.2. Hence, each row k can be computed from row $k-1$ using $O(\delta)$ binary intersections. In the case of T-INTERVAL-CONNECTIVITY, one simply has to repeat the operation until the T^{th}

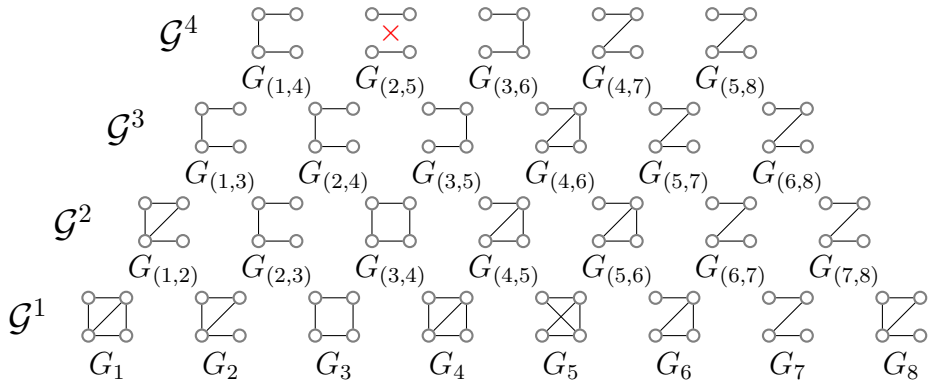


Figure 3.2: Example of an intersection hierarchy for a given dynamic graph $\mathcal{G}$ of length $\delta = 8$.

row, then answer `true` iff all graphs in this row are connected. The total cost is $O(\delta T) = O(\delta^2)$ binary intersections, plus $\delta - T + 1 = O(\delta)$ connectivity tests for the T^{th} row.

Solving INTERVAL-CONNECTIVITY is similar except that one needs to test the connectivity of all new graphs during the process. If a disconnected graph is first found in some row k, then the answer is $k-1$. If all graphs are connected up to row δ, then δ is the answer. Since there are $O(\delta^2)$ graphs in the intersection hierarchy, the total number of connectivity tests and binary intersections is $O(\delta^2)$.

Lower Bound. The following lower bound is valid for any algorithm that uses only the two elementary operations *binary intersection* and *connectivity test*.

Lemma 3.1. $\Omega(\delta)$ *elementary operations are necessary to solve* T-INTERVAL-CONNECTIVITY.

Proof. *(by contradiction).* Let $\mathcal{A}$ be an algorithm that uses only elementary operations and that decides whether any sequence of graphs is T-interval connected in $o(\delta)$ operations. Then, for any sequence $\mathcal{G}$, at least one graph in $\mathcal{G}$ is never accessed by $\mathcal{A}$. Let $\mathcal{G}_1$ be a sequence that is T-interval connected and suppose that $\mathcal{A}$ decides that $\mathcal{G}_1$ is T-interval connected without accessing graph G_k. Now, consider a sequence $\mathcal{G}_2$ that is identical to $\mathcal{G}_1$ except G_k is replaced by a disconnected graph G'_k. Since G'_k is never accessed, the executions of $\mathcal{A}$ on $\mathcal{G}_1$ and $\mathcal{G}_2$ are identical and $\mathcal{A}$ incorrectly decides that $\mathcal{G}_2$ is T-interval connected. □

A similar argument can be used for INTERVAL-CONNECTIVITY by making the answer T dependent on the graph G_k that is never accessed.

From the naive strategy, one can see that the computation of some intersection graphs is not required. Using the same example of the intersection graph $G_{(3,6)} =$

$G_3 \cap G_4 \cap G_5 \cap G_6$, we see that to compute $G_{(3,6)}$, we do not really need $G_{(3,5)}$ and $G_{(4,6)}$. One can simply compute it from $G_{(3,4)} \cap G_{(5,6)}$ whose result covers the whole sequence $\{G_3, G_4, G_5, G_6\}$. This can significantly reduce the amount of computation and gives the idea for a new row-based strategy.

3.2 Row-Based Strategy

In this section, we present a basic strategy that improves upon the previous naive strategy, yielding upper bounds of $O(\delta \log \delta)$ operations for both problems. Its main interest is in the fact that it can be parallelized, and this allows us to show that both problems are in **NC**, i.e. parallelizable on a PRAM with a polylogarithmic running time. We first describe the algorithms for a sequential machine (RAM).

Informally, the general strategy is to compute only some of the rows of $\mathcal{G}$'s intersection hierarchy as each intersection graph requires two graphs to be computed, it is sufficient that these two graphs cover the entire corresponding sequence. As we follow a row based strategy, the two graphs should be on the same row (we will see later that it is not necessary, and it will give a strategy for the optimal solution). The computation of a graph on the T^{th} row requires the two used graphs to be in a row $\mathcal{G}^i$ with $i \geq T/2$ as the height of an intersection graph equals the length of the sequence that it covers in the graph $\mathcal{G}$. The two graphs are computed in the same manner based on 4 graphs in the row $\mathcal{G}^{i/2}$ and so on. Following this principle it is possible, starting from the graph $\mathcal{G}$ in the base of the intersection hierarchy, to compute only necessary rows, namely $\mathcal{G}^{2^i}$ ("power rows"). Figure 3.3 shows the necessary computation to compute the intersection graph $\mathcal{G}^8[5] = \cap\{G_5, G_6, G_7, G_8, G_9, G_{10}, G_{11}, G_{12}\}$. The figure represents the intersection hierarchy of the graph $\mathcal{G}$ of length $\delta = 16$ (the base). Nodes colored in gray represent computed power rows graphs and the initial graph sequence.

Formally, the computation of "power rows" is based on the following lemma:

Lemma 3.2. *If some row $\mathcal{G}^k$ is already computed, then any row $\mathcal{G}^\ell$ for $k+1 \leq \ell \leq 2k$ can be computed with $O(\delta)$ elementary operations.*

Proof. Assume that row $\mathcal{G}^k$ is already computed and that one wants to compute row $\mathcal{G}^\ell$ for some $k+1 \leq \ell \leq 2k$. Note that row $\mathcal{G}^\ell$ consists of the entries $\mathcal{G}^\ell[1], \ldots, \mathcal{G}^\ell[\delta - \ell + 1]$. Now, observe that for any $k+1 \leq \ell \leq 2k$ and for any $1 \leq i \leq \delta - \ell + 1$, $\mathcal{G}^\ell[i] = G_{(i,i+\ell-1)} = G_{(i,i+k-1)} \cap G_{(i+\ell-k,i+\ell-1)} = \mathcal{G}^k[i] \cap \mathcal{G}^k[i+\ell-k]$. Hence, $\delta - \ell + 1 = O(\delta)$ intersections are sufficient to compute all of the entries of row $\mathcal{G}^\ell$. □

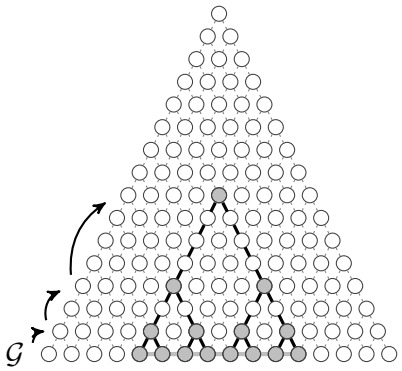


Figure 3.3: Example of computation of the intersection graph $\mathcal{G}^8[5]$ corresponding to the sequence $\{G_5, G_6, G_7, G_8, G_9, G_{10}, G_{11}, G_{12}\}$ based on power rows graphs.

T-INTERVAL-CONNECTIVITY

Using Lemma 3.2, we can incrementally compute "power rows" $\mathcal{G}^{2^i}$ for all i from 1 to $\lceil \log_2 T \rceil - 1$ without computing the intermediate rows. Then, we compute row $\mathcal{G}^T$ directly from row $\mathcal{G}^{2^{\lceil \log_2 T \rceil - 1}}$ (again using Lemma 3.2). This way, we compute $\lceil \log_2 T \rceil = O(\log \delta)$ rows using $O(\delta \log \delta)$ intersections, after which we perform $O(\delta)$ connectivity tests.

In Figure 3.4 we show a T-interval testing based on power rows computation of the graph $\mathcal{G}$ of length $\delta = 16$ with $T = 11$. Red (dark) nodes represent the graphs of the Tth row. The computation of the power rows stops upon reaching $\mathcal{G}^8$ which is the last power row before $\mathcal{G}^{11}$ (the first power row after $\mathcal{G}^{11/2}$). The graph of the Tth row are then computed from the intersection graph $\mathcal{G}^8$ using binary intersection according to Lemma 3.2, e.g $\mathcal{G}^{11}[2] = \mathcal{G}^8[2] \cap \mathcal{G}^8[5]$. Finally, T-interval connectivity is tested by testing the connectivity of each graph in $\mathcal{G}^{11}$.

INTERVAL-CONNECTIVITY

Here, we incrementally compute rows $\mathcal{G}^{2^i}$ until we find a row that contains a disconnected graph (thus, a connectivity test is performed after each intersection). By Lemma 3.2, each of these rows can be computed using $O(\delta)$ intersections. Suppose that row $\mathcal{G}^{2^{j+1}}$ is the first power row that contains a disconnected graph, and that $\mathcal{G}^{2^j}$ is the row computed before $\mathcal{G}^{2^{j+1}}$. Next, we do a binary search of the rows between $\mathcal{G}^{2^j}$ and $\mathcal{G}^{2^{j+1}}$: compute and test the middle row $\mathcal{G}^{2^j+2^{j-1}}$, then continue the same way between rows $\mathcal{G}^{2^j}$ and $\mathcal{G}^{2^j+2^{j-1}}$ if this latter contains at least one disconnected graph, otherwise, between $\mathcal{G}^{2^j+2^{j-1}}$ and $\mathcal{G}^{2^{j+1}}$. This is continued until we find the row $\mathcal{G}^T$ with the highest row number T such that all graphs on this row are connected (see Fig. 3.5 for an illustration of the algorithm). The computation of

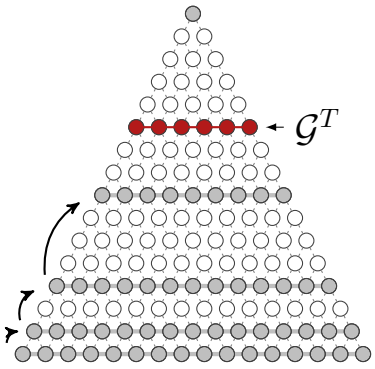


Figure 3.4: Example of T-interval connectivity testing based on the computation of power rows.

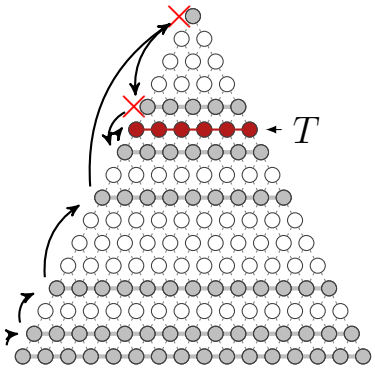


Figure 3.5: Example of interval connectivity testing based on the computation of power rows. Here $\delta = 16$ and $T = 11$.

each of these rows is based on row $\mathcal{G}^{2^j}$ and takes $O(\delta)$ intersections by Lemma 3.2. Overall, we compute at most $2\lceil \log_2 T \rceil = O(\log \delta)$ rows using $O(\delta \log \delta)$ intersections and the same number of connectivity tests.

Figure 3.5 shows an example of interval connectivity testing based on the computation of the power rows. In this example $\delta = 16$ and the found value of T is 11. The computation of power rows stops upon reaching $\mathcal{G}^{16}$ which contains a disconnected graph ($\times$). So necessarily $16 > T \geq 8$. A binary search between rows $\mathcal{G}^8$ and $\mathcal{G}^{16}$ is then used to find $\mathcal{G}^{11}$, the highest row where all graphs are connected.

3.2.1 Parallel Algorithm

In the row-based strategy we can see that the computation of each power row depends on the computation of the previous one which requires a logarithmic factor in the complexity. However, each computed graph is used in the construction of a constant number of graphs. This can be used to compute a linear number of graphs of the same row simultaneously using a linear number of processors. We now present a parallel version of the row-based algorithm on a EREW PRAM model. We specify that we always follow a high-level strategy, where the only parameter of the input size is δ (ignoring n and m).

Definition 3.6 (EREW PRAM)**.** *The Parallel Random Access Machine (PRAM) model is a generalization of the RAM model of sequential computation. It consists in a collection of numbered RAM processors $P_0, P_1, P_2, ...$, a collection of shared memory cells $M[0], M[1], M[2], ...$ Each P_i has its own local memory (registers) and knows its index i. In the Exclusive Read Exclusive Write (EREW) PRAM variant only one processor can read and write at a given time in a shared memory cell.*

Lemma 3.3. *If some row $\mathcal{G}^k$ is already computed, then any row between $\mathcal{G}^{k+1}$ and $\mathcal{G}^{2k}$ can be computed in $O(1)$ time on an EREW PRAM with $O(\delta)$ processors.*

Proof. Assume that row $\mathcal{G}^k$ is already computed, and that one wants to compute row $\mathcal{G}^\ell$, consisting of the entries $\mathcal{G}^\ell[1], \ldots, \mathcal{G}^\ell[\delta - \ell + 1]$, for some $k+1 \leq \ell \leq 2k$. Since $\mathcal{G}^\ell[i] = \mathcal{G}^k[i] \cap \mathcal{G}^k[i + \ell - k]$, $1 \leq i \leq \delta - \ell + 1$, the computation of row $\mathcal{G}^\ell$ can be implemented on an EREW PRAM with $\delta - \ell + 1$ processors in two rounds as follows. Let P_i, $1 \leq i \leq \delta - \ell + 1$, be the processor dedicated to computing $\mathcal{G}^\ell[i]$. In the first round P_i reads $\mathcal{G}^k[i]$, and in the second round P_i reads $\mathcal{G}^k[i + \ell - k]$. This guarantees that each P_i has exclusive access to the entries of row $\mathcal{G}^k$ that it needs for its computation. Hence, row $\mathcal{G}^\ell$ can be computed in $O(1)$ time on an EREW PRAM using $O(\delta)$ processors. □

Now we establish that T-INTERVAL-CONNECTIVITY and INTERVAL-CONNECTIVITY are in Nick's class (**NC**) by showing that our algorithms are efficiently parallelizable.

Time complexity on an EREW PRAM

***T*-INTERVAL-CONNECTIVITY on an EREW PRAM:** The sequential algorithm for this problem computes $O(\log \delta)$ rows. By Lemma 3.3, each of these rows can be computed in $O(1)$ time on an EREW PRAM with $O(\delta)$ processors. Therefore, all of the rows (and hence all necessary intersections) can be computed in $O(\log \delta)$ time with $O(\delta)$ processors. The $O(\delta)$ connectivity tests for row $\mathcal{G}^T$ can be done in $O(1)$ time with $O(\delta)$ processors. Then, the processors can establish whether or not all graphs in row $\mathcal{G}^T$ are connected by computing the logical AND of the results of the $O(\delta)$ connectivity tests in time $O(\log \delta)$ on an EREW PRAM with $O(\delta)$ processors using standard techniques (see Gibbons and Rytter [1988]; JáJá [1992]). The total time is $O(\log \delta)$ on an EREW PRAM with $O(\delta)$ processors.

INTERVAL-CONNECTIVITY on an EREW PRAM: The sequential algorithm for this problem computes $O(\log \delta)$ rows. Differently from T-INTERVAL-CONNECTIVITY, a connectivity test is done for each of the computed graphs (rather than just those of the last row) and it has to be determined for each computed row whether or not all of the graphs are connected. This takes $O(\log \delta)$ time for each of the $O(\log \delta)$ computed rows using the same techniques as for T-INTERVAL-CONNECTIVITY. The total time is $O(\log^2 \delta)$ on an EREW PRAM with $O(\delta)$ processors.

3.3 Optimal Solution

We now present our strategy for solving both T-INTERVAL-CONNECTIVITY and INTERVAL-CONNECTIVITY using a linear number of elementary operations (in the length δ of $\mathcal{G}$), matching the $\Omega(\delta)$ lower bound presented in Section 3.1.

To go from the row-based strategy to the optimal strategy we first note that two graphs used in the computation of an intersection graph can be on different rows in the hierarchy of intersection.

We show in Figure 3.6 an example of intersection graph $\mathcal{G}^{10}[3] = G_{(3,12)}$ computed from the two graphs $\mathcal{G}^6[3]$ that we call *left intermediate intersection graph* and $\mathcal{G}^4[9]$ called *right intermediate intersection graph* shown in dark gray that covers the entire corresponding sequence in $\mathcal{G}$ ($\mathcal{G}^{10}[3] = G_{(3,12)} = \mathcal{G}^4[9] \cap \mathcal{G}^6[3]$).

It is desirable that the already computed intermediate graphs can be reusable for computing other intermediate graphs or other graphs on the row that we want to compute. We show in Figure 3.6 the computation of the subsequent graph $\mathcal{G}^{10}[4] = G_{(4,13)}$ on row T. For an efficient use of intermediate graphs, two from these four graphs can be computed directly from the other two. According to the order of computation of the graphs in the row T, we can specify an order for com-

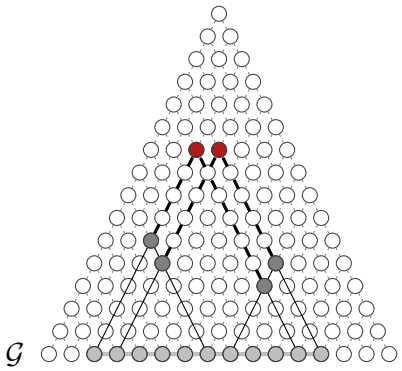


Figure 3.6: Example of computation of the intersection graphs $\mathcal{G}^{10}[3] = \cap\{G_3, G_4, \ldots, G_{12}\}$ and $\mathcal{G}^{10}[4] = \cap\{G_4, G_5, \ldots, G_{13}\}$ based on the computed graphs on different rows.

puting intermediate graphs. Let us consider the left to right direction. We use for the computation of $\mathcal{G}^{10}[3]$ to compute both graphs $\mathcal{G}^4[9]$ and $\mathcal{G}^6[3]$, the computation of the latter ($\mathcal{G}^6[3]$) requires the graph $\mathcal{G}^5[4]$ to be computed first. Then, for the computation of the graph $\mathcal{G}^{10}[4]$, it remains only to compute the intermediate graph $\mathcal{G}^5[9]$ directly from $\mathcal{G}^4[9] \cap G_{13}$. If we continue to walk right into row T, we see that for each computed graph, the right intermediate graph is computed directly from that of the previous graphs in the row, but the left intermediate graph can not be computed from that of the previous one but used to compute it, so it must be computed before. To make things clearer we introduce the notion of *ladder*. Informally, a *ladder* is a sequence of intermediate graphs that "climbs" the intersection hierarchy bottom-up.

Definition 3.7. *The* right ladder of length l at index i*, denoted by* $\mathcal{R}^l[i]$*, is the sequence of intersection graphs* $(\mathcal{G}^k[i],\ k = 1, 2, \ldots, l)$.
The left ladder of length l at index i*, denoted by* $\mathcal{L}^l[i]$*, is the sequence* $(\mathcal{G}^k[i - k + 1],\ k = 1, 2, \ldots, l)$.
A right (resp. left) ladder of length $l - 1$ *at index* i *is said to be* incremented *when graph* $\mathcal{G}^l[i]$ *(resp.* $\mathcal{G}^l[i - l + 1]$*) is added to it, and the resulting sequence of intersection graphs is called the* increment *of that ladder.*

Lemma 3.4. *A ladder of length* l *can be computed using* $l - 1$ *binary intersections.*

Proof. Consider a right ladder $\mathcal{R}^l[i]$. For any $k \in [2, l]$ it holds that $\mathcal{G}^k[i] = \mathcal{G}^{k-1}[i] \cap G_{i+k-1}$. Indeed, by definition, $\mathcal{G}^{k-1}[i] = \cap\{G_i, G_{i+1}, ..., G_{i+k-2}\}$. The ladder can thus be built bottom-up using a single new intersection at each level.

Consider a left ladder $\mathcal{L}^l[i]$. For any $k \in [2, l]$ it holds that $\mathcal{G}^k[i - k + 1] = G_{i-k+1} \cap \mathcal{G}^{k-1}[i - k + 2]$. Indeed, by definition, $\mathcal{G}^{k-1}[i - k + 2] = \cap\{G_{i-k+2},$

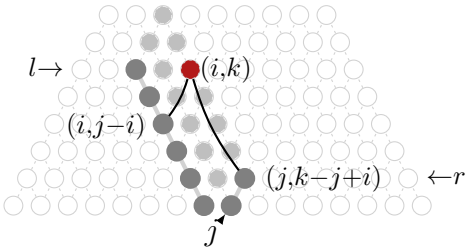


Figure 3.7: Examples of intersection rectangle computation based on left and right ladders.

$G_{i-k+3}, ..., G_i\}$. The ladder can thus be built bottom-up using a single new intersection at each level. □

Definition 3.8 (Intersection rectangle)**.** *Given $\mathcal{L}^l[j-1]$ and $\mathcal{R}^r[j]$, the set of graphs $\mathcal{G}^k[i]$ such that $j-l \leq i < j$ and $j-i < k \leq j-i+r$, forms an intersection rectangle defined by the triplet (j, l, r) delimited by the two ladders and the two lines that are parallel to the two ladders as shown in Figure 3.7. The graphs $\mathcal{G}^k[i]$ defined by the constraints above, shown in light gray in the figure, include all graphs that are strictly inside the rectangle, and all graphs on the parallel lines. Notice that graphs on the two ladders are excluded.*

Lemma 3.5. *Given a pair (i, k), the graph $\mathcal{G}^k[i]$ can be computed as $\mathcal{G}^x[i] \cap \mathcal{G}^{k-x}[i+x]$ such that $1 \leq x \leq k$.*

Proof. By definition, $\mathcal{G}^k[i] = \cap\{G_i, G_{i+1}, ..., G_{i+k-1}\}$ and $\mathcal{G}^x[i] = \cap\{G_i, G_{i+1}, ..., G_{i+x-1}\}$ and $\mathcal{G}^{k-x}[i+x] = \cap\{G_{i+x}, G_{i+x+1}, ..., G_{i+k-1}\}$ ($G_{i+x+k-x-1} = G_{i+k-1}$). It follows that $\mathcal{G}^k[i] = \mathcal{G}^x[i] \cap \mathcal{G}^{k-x}[i+x]$. □

Corollary 3.1. *Given an intersection rectangle (j, l, r), any graph $\mathcal{G}^k[i]$ in the intersection rectangle can be computed by a single binary intersection, namely $\mathcal{G}^k[i] = \mathcal{G}^{j-i}[i] \cap \mathcal{G}^{k-j+i}[j]$.*

Proof. From Lemma 3.5, with $j - i = x$, we have $\mathcal{G}^k[i] = \mathcal{G}^{j-i}[i] \cap \mathcal{G}^{k-j+i}[j]$. By definition, $\mathcal{G}^{j-i}[i] \in \mathcal{L}^l[j-1]$ and $\mathcal{G}^{k-j+i}[j] \in \mathcal{R}^r[j]$, so only a single binary intersection is needed. □

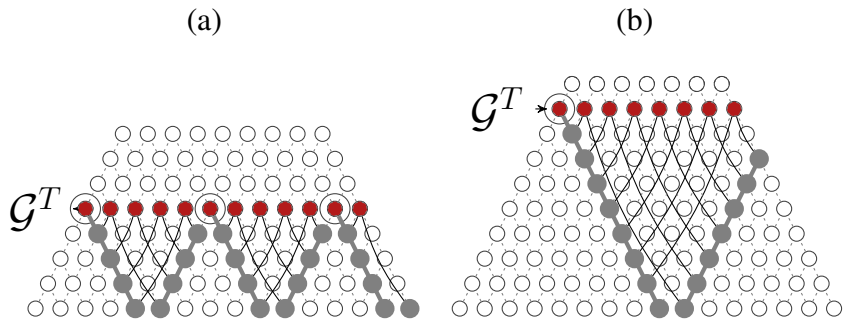


Figure 3.8: Examples of the execution of the optimal algorithm for T-INTERVAL-CONNECTIVITY with $T < \delta/2$ (a) and $T \geq \delta/2$ (b). $\mathcal{G}$ is T-interval connected in both examples.

T-INTERVAL-CONNECTIVITY

We describe our optimal algorithm for this problem with reference to Fig. 3.8 below which shows two examples of the execution of the algorithm (see Algorithm 2 for details). The algorithm traverses the T^{th} row in the intersection hierarchy from left to right, starting at $\mathcal{G}^T[1]$. If a disconnected graph is found, the algorithm returns `false` and terminates (line 14). If the algorithm reaches the last graph in the row, i.e. $\mathcal{G}^T[\delta - T + 1]$, and no disconnected graph was found, then it returns `true` (line 16). The graphs $\mathcal{G}^T[1], \mathcal{G}^T[2], \ldots, \mathcal{G}^T[\delta - T + 1]$ are computed based on the set of ladders $\mathcal{S} = \{\mathcal{L}^T[T], \mathcal{R}^{T-1}[T+1], \mathcal{L}^T[2T], \mathcal{R}^{T-1}[2T+1], \ldots\}$, which are constructed as follows. Each left ladder is built entirely (from bottom to top, line function `computeLeftLadder`) when the traversal arrives at its top location in row T (i.e. where the last increment is to take place, variable `next`). For instance, $\mathcal{L}^T[T]$ is built when the walk (current intersection graph) is at index 1 in row T, $\mathcal{L}^T[2T]$ is built at index $T + 1$, and so on. If a disconnected graph is found in the process, the execution terminates returning `false`.

```
1  k ← T                                                    // current row (non-changing)
2  i ← 1                                                    // current index in the row
3  next ← 1                                                 // trigger for next ladder construction
4  // walk until stepping out of the intersection hierarchy
5  while i ≤ δ − k + 1 do
6      if i = next then
7          next ← i + k
8                                                           //compute L^k[next − 1]
9          if ¬computeLeftLadder(k, next) then
10             return false
11     else
12                                                          //compute G^k[i] and G^{k−next+i}[next]
           computeFromIntersection(k, i, next)
13     if ¬isConnected(G^k[i]) then
14         return false
15     i ← i + 1
16 return true
```

```
17 function computeLeftLadder(k, next) :        // compute the left ladder L^k[next − 1]
18     k' ← 1                                                // row of first increment
19     i' ← next − 1                                         // index of first increment
20     while k' < k do
21         if ¬isConnected(G^{k'}[i']) then
22             return false                                  // a disconnected graph was found
23         k' ← k' + 1
24         i' ← i' − 1
25         G^{k'}[i'] ← G^{k'−1}[i' + 1] ∩ G_{i'}             // "increment" the ladder
```

```
26 function computeFromIntersection(k, i, next) :   // "increment" the right ladder
27     k' ← k − next + i                  // row of increment (right ladder), always k' > 1
28     incrementRightLadder(k', i, next)
```

```
29 function incrementRightLadder(k', i, next) :     // "increment" the right ladder
30     G^{k'}[next] ← G^{k'−1}[next] ∩ G_{next+k'−1}          // "increment" right ladder
31     G^k[i] ← G^{next−i}[i] ∩ G^{k'}[next]          // compute intersection based on Corollary 3.1
```

Algorithm 2: Optimal algorithm for T-INTERVAL-CONNECTIVITY

Differently from left ladders, right ladders are constructed gradually as the traversal proceeds. Each time that the traversal moves right to a new index in the T^{th} row, the current right ladder is incremented and the new top element of this right ladder is used immediately to compute the graph at the current index in the T^{th} row (using Corollary 3.1). This continues until the right ladder reaches row $T-1$ after which a new left ladder is built.

Theorem 3.1. *T-INTERVAL-CONNECTIVITY can be solved with $\Theta(\delta)$ elementary operations, which is optimal (to within a constant factor).*

Proof. The set $\mathcal{S}$ of ladders constructed by the algorithm includes at most $\lceil \delta/T \rceil$ left ladders and $\lceil \delta/T \rceil$ right ladders, each of length at most T. By Lemma 3.4, the set of ladders $\mathcal{S}$ can be computed using less than 2δ binary intersections. Based on Corollary 3.1, each of the $\delta - T + 1$ graphs $\mathcal{G}^T[i]$ in row T can be computed at the cost of a single intersection of two graphs in $\mathcal{S}$. At most $\delta - T + 1$ connectivity tests are performed for row T. This establishes the following result which matches the lower bound of Lemma 3.1. □

INTERVAL-CONNECTIVITY

The strategy of our optimal algorithm for this problem is in the same spirit as the one for T-INTERVAL-CONNECTIVITY. However, it is more complex and corresponds to a walk in the two dimensions of the intersection hierarchy. It is best understood with reference to Fig. 3.9 which shows an example of the execution of the algorithm (see Algorithm 3 for details).

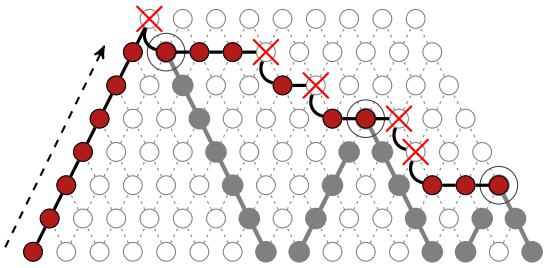


Figure 3.9: Example of the execution of the optimal algorithm for INTERVAL-CONNECTIVITY. *(It is a coincidence that the rightmost ladder matches the outer face.)*

The walk starts at the graph $\mathcal{G}^1[1]$ and builds a right ladder incrementally until it encounters a disconnected graph (first loop, line 5). If $\mathcal{G}^\delta[1]$ is reached and is connected, then $\mathcal{G}$ is δ-interval connected and the execution terminates returning δ. Otherwise, suppose that a disconnected graph is first found in row $k+1$. Then

```
1  k ← 1                                                          // current row
2  i ← 1                                                          // current index in the row
3  next ← 2                                                       // trigger for next ladder construction
4  // builds a right ladder until a disconnected graph is found
5  while isConnected(G^k[1]) do
6      k ← k + 1
7      if k > δ then
8          return δ                                               // the graph is δ-interval connected
9      else
10         G^k[1] ← G^{k−1}[1] ∩ G_k                              // "increment" the right ladder
11 if k = 1 then
12     return 0                                                   // the graph is 0-interval connected
13 k ← k − 1                                                      // move down
14 i ← i + 1                                                      // move right
15 // walk until stepping out of the hierarchy
16 while i ≤ δ − k + 1 do
17     if i = next then
18         next ← i + k
19         computeLeftLadder(k, i, next)
20     else
21         computeFromIntersection(k, i, next)
22         if ¬isConnected(G^k[i]) then
23             k ← k − 1
24     if k = 0 then
25         return 0
26     i ← i + 1
27 return k
```

```
28 function computeLeftLadder(k, i, next) :                       // compute the left ladder L^k[next − 1]
29     k' ← 1                                                     // row of first increment
30     i' ← next − 1                                              // index of first increment
31     while k' < k do
32         if ¬isConnected(G^{k'}[i']) then
33             k ← k' − 1                                         // move the original walk..
34             i ← i' + 1                                         // ..below-right disconnected graph,
35             return                                             // abort function
36         k' ← k' + 1
37         i' ← i' − 1
38         G^{k'}[i'] ← G^{k'−1}[i' + 1] ∩ G_{i'}                  // "increment" the ladder
```

```
39 function computeFromIntersection(k, i, next) :                 // "increment" the right ladder
   (same function as for Algorithm 2)
40     k' ← k − next + i                                          // row of increment (right ladder), always k' > 1
41     incrementRightLadder(k', i, next)
```

```
42 function incrementRightLadder(k', i, next) :                   // "increment" the right ladder (same
   function as for Algorithm 2)
43     G^{k'}[next] ← G^{k'−1}[next] ∩ G_{next+k'−1}              // "increment" right ladder
44     G^k[i] ← G^{next−i}[i] ∩ G^{k'}[next]                      // compute intersection based on Corollary 3.1
```

Algorithm 3: Optimal algorithm for INTERVAL-CONNECTIVITY

k is an upper bound on the connectivity of $\mathcal{G}$ and the walk drops down a level to $\mathcal{G}^k[2]$ which is the next graph in row k that needs to be checked. This requires the construction of a left ladder $\mathcal{L}^k[k+1]$ of length k ending at $\mathcal{G}^k[2]$ (function `coputeFromRight`). The walk proceeds rightward on row k using a similar traversal strategy to the algorithm for T-INTERVAL-CONNECTIVITY. Here, however, every time that a disconnected graph is found, the walk drops down by one row. The dropping down operation, say, from some $\mathcal{G}^k[i]$, is made in two steps (curved line in Fig. 3.9). First it goes to $\mathcal{G}^{k-1}[i]$, which is necessarily connected because $\mathcal{G}^k[i-1]$ is connected (so a connectivity test is not needed here), and then it moves one unit right to $\mathcal{G}^{k-1}[i+1]$ (line 22). If the walk eventually reaches the rightmost graph of some row and this graph is connected, then the algorithm terminates returning the corresponding row number as T. Otherwise the walk will terminate at a disconnected graph in row 1 and $\mathcal{G}$ is not T-interval connected for any T. In this case, the algorithm returns $T = 0$.

Similarly to the algorithm for T-INTERVAL-CONNECTIVITY, the computations of the graphs in a walk by Algorithm 3 (for INTERVAL-CONNECTIVITY) use binary intersections based on Lemma 3.4 and Corollary 3.1. If the algorithm returns that $\mathcal{G}$ is T-interval connected, then each graph $\mathcal{G}^T[1], \mathcal{G}^T[2], \ldots, \mathcal{G}^T[\delta - T + 1]$ must be connected. The graphs that are on the walk are checked directly by the algorithm.

Lemma 3.6. *If an intersection graph $G_{(i,j)}$ is connected then any graph $G_{(i',j')}$ such that $i' \geq i$ and $j' \leq j$ is connected.*

Proof. By definition $G_{(i,j)}$ is connected implies that the sequence $\{G_i, G_{i+1}, ..., G_j\}$ shares a common connected spanning subgraph. As the sequence $\{G_{i'}, G_{i'+1}, ..., G_{j'}\}$ is a subsequence of $\{G_i, G_{i+1}, ..., G_j\}$, then it shares the same common connected spanning subgraph. □.

According to Lemma 3.6, for each graph $\mathcal{G}^T[i]$ on row T that is below the walk, there is a graph $\mathcal{G}^j[i]$ with $j \geq T$ that is on the walk and is connected and this implies that $\mathcal{G}^T[i]$ is connected.

Theorem 3.2. INTERVAL-CONNECTIVITY *can be solved with $\Theta(\delta)$ elementary operations, which is optimal (up to a constant factor).*

Proof. The ranges of the indices covered by the left ladders that are constructed by the process are disjoint, so their total length is O(δ). The first right ladder has length at most δ and each subsequent right ladder has length less than the left ladder that precedes it so the total length of the right ladders is also O(δ). Therefore, this algorithm performs at most O(δ) binary intersections and O(δ) connectivity tests. This establishes that INTERVAL-CONNECTIVITY can be solved with $\Theta(\delta)$ elementary operations, which matches the lower bound of Lemma 3.1. □

3.3.1 Online Algorithms

The optimal algorithms for T-INTERVAL-CONNECTIVITY and INTERVAL-CONNECTIVITY can be adapted to an online setting in which the sequence of graphs $G_1, G_2, G_3, \ldots$ of a dynamic graph $\mathcal{G}$ is processed in the order that the graphs are received. In the case of T-INTERVAL-CONNECTIVITY, the algorithm cannot provide an answer until at least T graphs have been received. When the T^{th} graph is received, the algorithm builds the first left ladder using $T-1$ binary intersections. It can then perform a connectivity test and answer whether or not the sequence is T-interval connected so far. After this initial period, a T-connectivity test can be performed for the T most recently received graphs (by performing a connectivity test on the corresponding graph in row T) after the receipt of each new graph.

Theorem 3.3. *T-INTERVAL-CONNECTIVITY and INTERVAL-CONNECTIVITY can be solved online with an amortized cost of $\Theta(1)$ elementary operations per graph received.*

Proof. At no time does the number of intersections performed to build left ladders exceed the number of graphs received and the same is true for right ladders. Furthermore, each new graph after the first $T-1$ corresponds to a graph in row T which can be computed with one intersection by Corollary 3.1. In summary, the amortized cost is $O(1)$ elementary operations for each graph received and for each T-connectivity test after the initial period. The analysis for INTERVAL-CONNECTIVITY is similar except the algorithm can report the connectedness of the sequence so far starting with the first graph received. □

By adapting our strategy directly to an online version, a graph whose prefix is not T-interval connected will never be T'-interval connected for $T' > T$. If we take the T-INTERVAL-CONNECTIVITY as a form of stability, it would be interesting therefore to consider only the recent state of the graph. Considering a dynamic online version allows to characterize the stability during the last term.

3.4 Dynamic Online Interval Connectivity

The algorithms in this section are motivated by Internet protocols like TCP (Transmission Control Protocol) which adjust their behaviour dynamically in response to recent network events and conditions such as dropped packets and congestion. T-interval connectivity is a measure of the stability of a network. Generally, larger values of T indicate that communication is more reliable, so it is natural to consider a dynamic version of interval connectivity that is based only on the recent states of a network rather than the entire history of a network. We formalize this notion of

recent history by introducing the concept of T-*stable* graphs. We then define the dynamic online versions of both T-INTERVAL-CONNECTIVITY and INTERVAL-CONNECTIVITY in terms of T-stable graphs.

Definition 3.9 (***T***-stable graph)**.** *A graph G_i, $i \geq T$, of a sequence $\mathcal{G} = \{G_1, G_2, ..., G_\delta\}$ is* T-stable *for a given T iff the subsequence $G_{i-(T-1)}, G_{i-(T-2)}, \ldots, G_{i-1}, G_i$ is T-interval connected.*

Definition 3.10 (Testing ***T***-STABILITY)**.** *The* T-STABILITY *problem for a given T is the problem of deciding for each received graph G_i, $i \geq T$, whether G_i is T-stable.*

Definition 3.11 (Testing STABILITY)**.** *We use the term* STABILITY *to refer to the problem of finding $T_i = \max\{T : G_i \text{ is } T\text{-stable}\}$ for each received graph G_i.*

As before, the first problem is a decision problem with true/false output, while the second is a maximization problem with integer output. Here, however, one such output is required after each graph in the sequence is received.

T-STABILITY

Our algorithm for T-STABILITY is similar to Algorithm 2 for T-INTERVAL-CONNECTIVITY. The differences are that the algorithm for T-STABILITY produces an output after each graph of a sequence is received, and the algorithm does not terminate if a disconnected graph is found on row T of the hierarchy. Instead, it continues until the last graph in the sequence is received. The ladders constructed by the algorithm for T-STABILITY are the same as the ladders that would be constructed by Algorithm 2 for a dynamic graph that is T-interval connected (see Figure 3.8 for examples). Given a dynamic graph $\mathcal{G} = \{G_1, G_2, ..., G_\delta\}$, T-STABILITY is undefined for the graphs G_i with $i < T$, so the algorithm returns $\perp$ after each of the first $T - 1$ graphs is received. When G_T is received, the algorithm builds a left ladder and returns `true` (resp. `false`) if the top graph of the ladder (i.e. $\mathcal{G}^T[1]$) is connected (resp. disconnected). Then the walk progresses rightward along row T every time that a graph is received, alternately building left and right ladders in such a way that the graph $\mathcal{G}^T[i-(T-1)]$ can always be computed from G_i with a single intersection (using Corollary 3.1). G_i is T-stable iff $\mathcal{G}^T[i-(T-1)]$ is connected and `true` or `false` is output as appropriate.

Theorem 3.4. T-STABILITY *can be solved online with an amortized cost of $\Theta(1)$ elementary operations per graph received.*

Proof. By the same analysis as the analysis for the online version of Algorithm 2, the number of intersections performed to build left ladders never exceeds the number of graphs received, and the same is true for the number of intersections to build right ladders and the number of connectivity tests. □

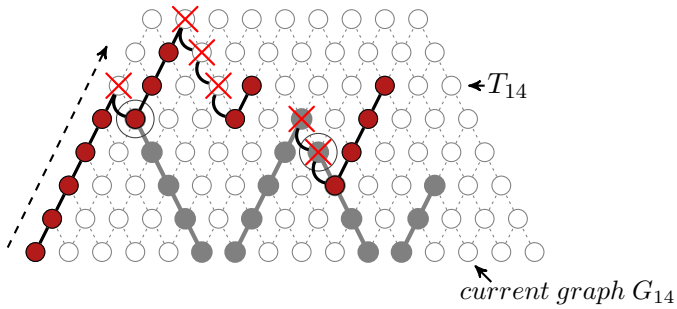


Figure 3.10: Example of the execution of the STABILITY algorithm.

STABILITY

The algorithm for this problem must find $T_i = \max\{T : G_i \text{ is } T\text{-stable}\}$ for each received graph G_i. Our algorithm for STABILITY generalizes the strategy that we used in the algorithm for INTERVAL-CONNECTIVITY by trying to climb as high as possible in the hierarchy, even after a disconnected intersection graph is found. This is necessary because the sequence of values $T_1, T_2, T_3, \ldots$ for STABILITY is not necessarily monotonic.

The algorithm for STABILITY uses right and left ladders to walk through the intersection hierarchy. The general idea is that the walk goes up when the current intersection graph is connected and down when it is disconnected (unless the walk is on the bottom level of the hierarchy in which case it goes right to the next graph). This is different from the algorithm for INTERVAL-CONNECTIVITY which only goes up during the construction of the first right ladder and goes right or down in all other cases. We will describe the algorithm for STABILITY with reference to Fig. 3.10 which shows an example of the execution of the algorithm. See Algorithm 4 for complete details.

The walk begins by constructing a right ladder. In each step, if a computed intersection graph $\mathcal{G}^k[j]$ is connected, and G_i, $i = j + k - 1$, is the most recently received graph, then the value k is returned to indicate that G_i is k-stable. Then the walk climbs one row in the hierarchy to $\mathcal{G}^{k+1}[j]$ which takes into consideration the next graph G_{i+1}. If a computed intersection graph $\mathcal{G}^k[j]$, $k > 1$, is disconnected, then the walk descends to the next graph in the row below, i.e. to $\mathcal{G}^{k-1}[j+1]$. In this case no value is returned because the next graph in $\mathcal{G}$ has not yet been considered. If a graph $\mathcal{G}^1[j]$ is disconnected, then 0 is returned, and the walk moves right to the next graph.

As in the previous algorithms, the right ladders are constructed incrementally as the walk goes up, even though each graph $\mathcal{G}^k[j]$ can be computed from $\mathcal{G}^{k-1}[j] \cap G_{j+k-1}$, because this prepares the ladders needed to compute the intersection graphs

if the walk goes down. This is illustrated by the second right ladder $\mathcal{R}^5[7]$ in Fig. 3.10. If a disconnected graph $\mathcal{G}^k[j]$ is found while building a right ladder, the walk jumps to the next graph in the row just below, i.e. to $\mathcal{G}^{k-1}[j+1]$, to avoid unnecessary computations. For example, in Fig. 3.10 the walk jumps from $\mathcal{G}^5[7]$ which is disconnected to $\mathcal{G}^4[8]$ without computing $\mathcal{G}^7[5]$ and $\mathcal{G}^6[6]$.

If a graph $\mathcal{G}^k[j]$ cannot be computed using the current ladders, then a complete new left ladder $\mathcal{L}^k[k+j-1]$ is constructed as high as possible until it reaches a previously computed graph or until it encounters a disconnected graph. The former case is illustrated by the left ladder $\mathcal{L}^5[6]$ in Fig. 3.10 which is built when the walk descends from $\mathcal{G}^6[1]$ to $\mathcal{G}^5[2]$. The latter case is illustrated by the left ladder $\mathcal{L}^4[11]$ which encounters the disconnected graph $\mathcal{G}^4[8]$. In this case the walk resumes from the previous graph in the ladder ($\mathcal{G}^3[9]$ in the example). In contrast, a new left ladder is not needed when the walk descends three times from $\mathcal{G}^8[2]$ to $\mathcal{G}^5[5]$ because the ladders $\mathcal{L}^5[6]$ and $\mathcal{R}^3[7]$ that exist at this point can be used to compute these intersections.

In the example in Fig. 3.10, the sequence of values $T_1, T_2, T_3, \ldots, T_{14}$ that the algorithm outputs is $1, 2, 3, 4, 5, 5, 6, 7, 5, 6, 3, 4, 5, 6$.

Theorem 3.5. STABILITY *can be solved with an amortized cost of* $\Theta(1)$ *elementary operations per graph received.*

Proof. The complexity analysis of the algorithm is similar to the analysis of the online algorithm for INTERVAL-CONNECTIVITY. The number of intersection graphs in right ladders never exceeds the number of graphs received and the same is true for left ladders. Each intersection graph in a ladder is computed using one binary intersection operation. Each time the walk climbs in the intersection hierarchy, one connectivity test is performed and a single graph is processed. When the walk descends, a new graph in $\mathcal{G}$ is not processed, but the number of descents cannot exceed the number of ascents, and each descent uses at most one connectivity test. This results in a constant amortized cost for each received graph. □

Conclusion

In this chapter, we studied the problem of testing whether a given dynamic graph $\mathcal{G} = \{G_1, G_2, ..., G_\delta\}$ is T-interval connected. We also considered the related problem of finding the largest T for which a given $\mathcal{G}$ is T-interval connected. We investigated algorithmic solutions that use two elementary operations, *binary intersection* and *connectivity testing*, to solve the problems. We developed efficient algorithms that use only $O(\delta)$ elementary operations, asymptotically matching the lower bound of $\Omega(\delta)$. We presented PRAM algorithms that show that both problems

```
1  k ← 1
2  i ← 1                                                    // current index in the row
3  next ← 2                                                 // trigger for next ladder construction
4  output ← 0

5                                                           // until the last graph is received
6  while receiving graphs do
7      while isConnected(G^k[i]) do
8          output ← k; k ← k + 1
9          computeFromIntersection(k, i, next)              // "increment" the right ladder
10     while ¬isConnected(G^k[i]) do
11         if k = 1 then
12             output ← 0; next ← i + 2
13         else
14             k ← k − 1
15         i ← i + 1
16         if i = next then
17             next ← i + k; computeLeftLadder(k, i, next)
18         else
19             computeFromIntersection(k, i, next)

20 function computeLeftLadder(k, i, next) :                 // compute the left ladder L^k[i]
21     k' ← 1                                               // row of first increment
22     i' ← next − 1                                        // index of first increment
23     while k' < k do
24         if ¬isConnected(G^{k'}[i']) then
25             k ← k' − 1                                   // move the original walk..
26             i ← i' + 1                                   // ..below-right disconnected graph,
27             return                                       // abort function
28         k' ← k' + 1; i' ← i' − 1
29         G^{k'}[i'] ← G^{k'−1}[i' + 1] ∩ G_{i'}           // "increment" the ladder

30 function computeFromIntersection(k, i, next) :
31     if i = next − 1 then
32         G^k[i] ← G^{k−1}[i] ∩ G_{i+k−1}
33     else
34         k' ← k − next + i                                // row of increment (right ladder)
35         G^{k'}[next] ← G^{k'−1}[next] ∩ G_{next+k'−1}    // "increment" right ladder
36         if ¬isConnected(G^{k'}[next]) then
37             i ← next; k ← k'
38         else
39             G^k[i] ← G^{next−i}[i] ∩ G^{k'}[next] // compute intersection based on Corollary 3.1
```

Algorithm 4: Optimal algorithm for STABILITY

can be solved efficiently in parallel, and online algorithms that use $\Theta(1)$ elementary operations per graph received. We also presented dynamic versions of the online algorithms that report connectivity based on recent network history.

Distributed algorithms for all of these problems, in which a node in the graph

only sees its local neighbourhood, would also be of interest. For example, distributed versions of the dynamic algorithms for T-INTERVAL-CONNECTIVITY in Section 3.4 could be used to supplement the information available to distributed Internet routing protocols such as OSPF (Open-Shortest Path First) which are used to construct routing tables. Our dynamic algorithms have $\Theta(1)$ amortized complexity, and distributed versions with $\Theta(1)$ amortized complexity could provide real-time information about network connectivity to OSPF.

Chapter 4

A Generic Framework for Testing Properties in Dynamic Graphs

Contents

In this chapter we propose a generic framework for testing properties in sequence-based dynamic graphs. This framework is a generalization of the framework presented in Chapter 3 to solve the problem of testing T-interval connectivity and that of finding the largest T for which the graph is T-interval connected. This work aims to formalize a testing process that allows to hide details related to low-level structures and focus on a high-level study of the problem.

4.1 Introduction

In Chapter 3 we presented a solution to the problem of testing the membership of a graph to the class T-interval connectivity for given T, and the related problem of finding the largest T for which the graph is T-interval connected.

The proposed framework was based on two elementary operations: *intersection* and *connectivity test*. These operations were used to walk through a hierarchy of intersection graphs and eventually answer the question. In this chapter, we present a generalization of this framework that allows one to test other classes and properties. Following the same principle, but using different operations (i.e. replacing *intersection* and *connectivity test* by other operations), other classification problems can be solved with the same high-level logic. Beyond the operations, we had reformulated the problem INTERVAL-CONNECTIVITY as a maximization problem, i.e. it consists in finding the *largest* value T corresponding to the highest row in the hierarchy for which the graph is T-interval connected. Some problems may however take the form of minimization problems.

In what follows, we will look at this kind of problems that unlike INTERVAL-CONNECTIVITY, consists in finding a minimum value corresponding to a lower level (row) in the hierarchy. For both kinds of problems (minimization and maximization problems) the optimal strategy presented for INTERVAL-CONNECTIVITY can be used and adapted to the problem (i.e the walk strategy and the use of ladders). In the case of maximization problems, the algorithm first searches a larger value by climbing the hierarchy as the test is positive. Then, the walk goes down the hierarchy in every step if the test is negative. In the case of a minimization problem, the walk moves up when the test is negative, otherwise it moves forward on the same row.

The chapter is organized as follows. Section 5.2 presents the model, basic definitions, and a generalization of the framework. In Section 4.3 we provide a generic algorithm for minimization problems. Then, in the remaining three sections we propose solutions based on three different problems using the presented generic algorithm. In Section 4.4, the framework is adapted to find the smallest duration for which all edges of the footprint appear, whatever the starting date (Class $\mathcal{B}$). In Section 4.5, we consider the problem of finding the (worst) *temporal diameter* of a given dynamic graph $\mathcal{G}$ over its lifetime. Finally, we investigate in Section 4.6 a somewhat more complex problem, that of computing the *round trip diameter* of a given graph $\mathcal{G}$.

4.2 Model and Definitions

In this chapter, we consider the same graph model as in Chapter 3. The dynamic graph is given as an untimed evolving graph. We assume that the changes between two consecutive graphs are arbitrary.

For a generalization of the framework presented in Chapter 3, we first define what is common in any use. We remind that the framework is based on two operations: a *composition* operation and a *test* operation. In the case of T-INTERVAL-CONNECTIVITY, the composition operation was *intersection* of two graphs and the test operation was *connectivity testing* of one graph. To generalize this concept, we use the notion of *super node*, i.e. a graph that is considered as a node in another graph. In Chapter 3 the super nodes were the graphs of the initial sequence $\{G_1, G_2, \dots\}$ as well as all the graphs which results from intersecting subsequences of these graphs (intersection hierarchy). To be more general, we will assume that a pre-processing can be made on the initial graphs $\{G_1, G_2, \dots\}$ giving as a result the sequence $\{G_{(i,i)}\}$ that may or may not be equal to $\{G_1, G_2, \dots\}$ (some of our examples uses such a pre-processing). Thus, the initial $G_i s$ are not super nodes themselves, but all $G_{(i,i)}$s are. We now define all these notions more precisely.

Definition 4.1 (Composition of super nodes)**.** *We call* composition operation *any binary operation that maps two graphs into another graph. Given such an operation* comp *and possibly some restrictions on its application (that depends upon the problem considered), we define the set of super nodes $S = \{G_{(i,j)} : i \leq j\}$ recursively by starting from the $\{G_{(i,i)}\}$s and adding all elements that can be obtained through composing two elements from S (a super node $G_{(i,j)}$ relates to the sequence $\{G_{(i,i)}, G_{(i+1,i+1)}, G_{(i+2,i+2)}, \dots, G_{(j,j)}\}$). In mathematical terms, S is the closure of $\{G_{(i,i)}\}$ under the composition operation (with restrictions), and the latter has for signature* comp*: $S \times S \to S$.*

An example of restriction that we will use, for one of the problem, is that if some $G_{(i,j)}$ is composed with some $G_{(i',j')}$, then i' must be equal to $j + 1$ (which is by chance the case with the high-level algorithm). In the cases that we will discuss, *super nodes* are graphs (they can take other forms in general). We say that the super node $G_{(i,j)}$ corresponds to the sequence $\{G_i, G_{i+1}, ...G_{j-1}, G_j\}$. We also use $\mathcal{G}^{j-i+1}[i]$ to denote the super node $G_{(i,j)}$

Definition 4.2 (Hierarchy)**.** *We call hierarchy the structure formed by the superposition of rows $\mathcal{G}^1, \mathcal{G}^2, \dots, \mathcal{G}^\delta$ where $\mathcal{G}^i = \{G_{(1,i)}, G_{(2,i+1)}, \dots, G_{(\delta-i+1,\delta)}\}$.*

Definition 4.3 (Test operation)**.** *The* test operation *is the elementary operation $test : S \to boolean$ that applies to a super node, and which allows the algorithm to determine in each step the next move (explained in the next section), until termination.*

4.3 Generic Framework

In order to reuse the proposed framework for other classes of dynamic graphs, the question must be reformulated as a minimization or maximization problem. Consider, for instance, the class studied in Chapter 2, *temporal connectivity*. As this class is defined by a general property on the entire graph without any parameter on it (as T in T-interval connectivity), it would not be significantly beneficial to use the framework to solve the decision problem of testing the membership of a dynamic graph to this class (an efficient algorithm is proposed in Chapter 2). However we can investigate the minimization problem of finding the *temporal diameter* d of a dynamic graph i.e. the smallest d such that in every subsequence of length d in the sequence $\mathcal{G}$, journeys exist from any node to all other nodes.

The related class *Round trip temporal connectivity* also could be interesting to study. In this case the aim would be to compute the *round trip diameter* rtd i.e. the smallest rtd such that in every subsequence of length rtd in the sequence $\mathcal{G}$, back-and-forth journeys exist from any node to all other nodes.

One among the most relevant classes to our framework is the class *Time-Bounded reappearance of edges* (Class $\mathcal{B}$). The problem that we can consider is that of finding the smallest bound b for which the graph has a *bounded realization of the footprint* i.e. the smallest b such that in every subsequence of length b in the sequence $\mathcal{G}$, all the edges of the footprint appear at least once.

The considered problem must be formulated in the context of our framework (in terms of rows and super nodes). In a minimization problem, the aim is to find the smallest parameter (as T in INTERVAL-CONNECTIVITY) for which the dynamic graph has the considered property. In our context, it comes to find the lowest row in the hierarchy where any super node satisfies the *test* operation.

In this section, we propose a generic algorithm. We will describe the algorithm at a high level using elementary operations (*composition* and *test* operations defined above). Then, the algorithm can be used directly, as we will show later, to solve specific minimization problems by defining the appropriate operations. We will not consider the decision variant of each problem since their algorithms can be deduced readily from those for the minimisation/maximization variant in general.

4.3.1 Generic Algorithm for Minimization Problems

To solve a given minimization problem, the generic algorithm for minimization problems follows almost the same principle as that for INTERVAL-CONNECTIVITY. The basics of the algorithm remain almost unchanged. An appropriate composition

operation is used to compute super nodes in the same way based on ladders also computed in the same manner. Only the walk is affected by the change of the purpose (finding the lowest row where any super node satisfies the test operation). In this case, the walk goes up in the hierarchy if the test is negative (instead of moving down as in the algorithm for INTERVAL-CONNECTIVITY), otherwise it moves forward on the same row. If the algorithm hits the right side of the hierarchy and the last visited super node $\mathcal{G}^k[\delta - k + 1]$ in the row $\mathcal{G}^k$ satisfies the test operation, then it terminates and returns the corresponding r. Otherwise, the walk climbs the right side of the hierarchy ($\mathcal{G}^{k+1}[\delta - k], \mathcal{G}^{k+2}[\delta - k - 1], \dots$) until the test is positive. If the walk reaches $\mathcal{G}^1[\delta]$ and the test is negative, then, the algorithm returns 0 indicating that the dynamic graph does not have the property. Algorithm 5 provides a high-level formal description. Figure 4.1 shows an example of execution of the generic algorithm for minimization problems. This figure will be used as a reference to describe the application of the algorithm to solve the minimization problems we will study.

Super nodes computation (Function `compute()`)**:** Intermediate super nodes (ladders) are computed the same way as in the algorithm for INTERVAL-CONNECTIVITY using the corresponding composition operation (instead of *intersection*). The super nodes resulting from the walk (red super nodes in Figure 4.1) are computed almost the same way. We describe computation (Function `compute()` in Algorithm 5) with reference to Figure 4.1. When the walk moves one step forward on the same row, the next super node is computed from the right and the left ladders ($\mathcal{G}^4[6] = comp(\mathcal{G}^2[6], \mathcal{G}^2[8])$) or from the ladder to which it belongs and a super node in the first row ($\mathcal{G}^4[4] = comp(\mathcal{G}^1[4], \mathcal{G}^3[5])$). If the walk climbs (moves up) a step in the hierarchy, then the next super node is computed from the preceding super node in the walk and the next super node in $\mathcal{G}^1$ (instead of using ladders), such as building a left ladder, ($\mathcal{G}^5[6] = comp(\mathcal{G}^4[6], \mathcal{G}^1[10])$). Although this computation does not require the use of ladders, the process continues to build a right ladder as the walk goes up for later use (if after-ward the walk moves forward on the same row, $\mathcal{G}^5[7]$). As it turns out, this generic algorithm for minimization problems, as well as its maximization analogue in Chapter 3, have a convenient property which will prove important for correctness for the last two problems (TEMPORAL-DIAMETER, ROUND-TRIP-TEMPORAL-DIAMETER).

Lemma 4.1 (Disjoint sequences property)**.** *If the algorithm performs a composition of two super nodes $G_{(i,j)}$ and $G_{(i',j')}$, then the corresponding sequences $\{G_i, G_{i+1}, \dots, G_j\}$ and $\{G'_i, G_{i'+1}, \dots, G_{j'}\}$ are disjoint and consecutive. That is, in any execution, $G_{(i,j')} = comp(G_{(i,j)}, G_{(i',j')}) \Rightarrow j = i' - 1$.*

Proof. According to the algorithm, any super node of the hierarchy is computed from: 1) two super nodes of two different ladders, a left one and a right one 2) a super node of a ladder and a super node in the first row 3) a super node of the

walk and a super node in the first row. In the three cases the two sequences covered by the two super nodes used are disjoint and consecutive: in any execution, $G_{(i,j')} = comp(G_{(i,j)}, G_{(i',j')}) \Rightarrow j = i' - 1$.

Theorem 4.1. *The generic algorithm for minimization problems has a cost of* $\Theta(\delta)$ *elementary operations.*

Proof. As the algorithm for INTERVAL-CONNECTIVITY, the total length of right and left ladders is O(δ). It was shown in Chapter 3 that any super node can be computed using a single composition operation based on ladders (Lemma 3.1). According to the algorithm, the number of super nodes computed by the walk is O(δ) and any computed super node is tested at most once. So, the generic algorithm for minimization problems performs O(δ) composition operations and O(δ) test operations. □

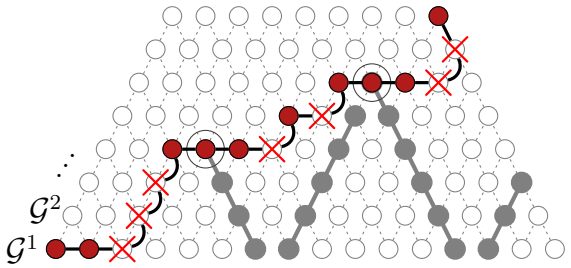


Figure 4.1: Example of the execution of the generic algorithm for minimization problems.

4.4 Bounded Realization of the Footprint

In this section, we illustrate the use of the generic framework in the context of Class $\mathcal{B}$ of those dynamic graphs with *Time-bounded reappearance of edges* (def 1.28). This class is defined by a property on edges recurrence over time in a dynamic graph. Let us first define an analogue of Class $\mathcal{B}$ in a *discrete* context (untimed evolving graphs). A graph has a time-bounded reappearance of edges with a bound b if the number of *steps* between two appearances of the same edge in the graph $\mathcal{G}$ is at most b. Formally:

Definition 4.4 (Class Time-bounded reappearance of edges (Class $\mathcal{B}$, Def. 1.28))**.** *Let* $\mathcal{G} = \{G_i = (V, E_i)\}$ *of length* δ *be a dynamic graph and* $G = (V, E)$ *its footprint,* $\mathcal{G}$ *has a time-bounded reappearance of edges within* b *iff* $\forall e \in E, \forall t \in$

```
1  i ← 1                                                  // current index in the row
2  k ← 1                                                  // current row

3  while k ≤ δ do
4  |  compute(G^k[i])
5  |  if test(G^k[i]) then
6  |  |  if i = δ − k + 1 then
7  |  |  |  return k                                     // return the number of the current row if
8  |  |  |                                               // the test on the right-most super node is true
9  |  |  else
10 |  |  |  i++                                          // move a step forward
11 |  else
12 |  |  if i = δ − k + 1 then
13 |  |  |  i−−; k++                                     // climb a step on the right side of the hierarchy
14 |  |  else
15 |  |  |  k++                                          // move up if the test on the super node is negative
16 return 0                                              // return 0 if the test on G^δ[1] is negative
```

Algorithm 5: Generic algorithm for minimization problems

$[1, \delta - b + 1], \exists t' \in [t, t + b - 1], e \in E_{t'}$. *It belongs to Class $\mathcal{B}$ if, in addition, the footprint G is connected. (In this discrete variant of $\mathcal{B}$ we ignore aspects related to communication latency on existing edges).*

We have seen in Chapter 1 that this property, together with the knowledge of n (the number of nodes) and b, allows the feasibility of shortest broadcast with termination detection (Casteigts *et al.* [2014]). Besides that, the value b determines a bound on the temporal diameter of the dynamic graph which is at most $b.Diameter(G)$.

Observation 4.1 (Bound on the footprint realization)**.** *The bound b of edges reappearance in a dynamic graph $\mathcal{G}$ defines a* bound on the footprint realization. *In other words in any period (subsequence) of length b in the graph $\mathcal{G}$, all the edges of the footprint appears.*

The problem we study in this section is to find the smallest bound on the footprint realization defined by the smallest reappearance bound of edges b in a dynamic graph $\mathcal{G}$. We can start by formulating a definition of the property based on our model and structure. The fact that there is a bound b on the footprint realization in a dynamic graph $\mathcal{G}$ implies that in any period b all edges of the footprint appear at least once. From our first definition of the property (Def. 4.4), we can say that b is a bound on the footprint realization if in each sequence of length b in the graph $\mathcal{G}$ all edges of the footprint appear at least once.

Definition 4.5 (BOUNDED-REALIZATION-OF-THE-FOOTPRINT)**.** *We will use* BOUNDED-REALIZATION-OF-THE-FOOTPRINT *to refer to the problem of finding* $\min\{b : \forall \mathcal{G}' = \{G_i, G_{i+1}, ..., G_{i+b-1}\} \subseteq \mathcal{G}, \forall e \in E(G), e \in \cup\{E_i, E_{i+1}, ..., E_{i+b-1}\}\}$ *for a given* $\mathcal{G}$*.*

Composition and test operations: The composition operation we will use is the *union* between two super nodes, which yields a union hierarchy. The test operation we will use is *equality to the footprint* (*equality* for short). The choice of these operations is explained next in the description of the algorithm.

Observation 4.2 (Cost of the operations)**.** *Using adjacency matrix, union operation and equality test can be done in* $O(n^2)$*. Note that since we address an offline analysis context, we consider that the same node in different graphs can be identified, thus isomorphism is not an issue.*

Lemma 4.2. *Let* $G_{(i,j)}$ *and* $G_{(i',j')}$ *be two super nodes and* $G_{(i,j')} = G_{(i,j)} \cup G_{(i',j')}$*.* $G_{(i,j)}$ *equals the footprint* $G \Rightarrow G_{(i,j')}$ *equals the footprint* G*.*

Proof. We have $G_{(i,j')} = G_{(i,j)} \cup G_{(i',j')} \Rightarrow G_{(i,j)} \subseteq G_{(i,j')}$. So, if $G_{(i,j)}$ equals the footprint G, then $G_{(i,j')}$ equals the footprint G. □

Observation 4.3. *Our minimization problem* BOUNDED-REALIZATION-OF-THE-FOOTPRINT *comes to finding the lowest row* $\mathcal{G}^b$ *where any super node* $\mathcal{G}^b[i]$ *for* $i \in [1, \delta - b + 1]$ *equals the footprint* G*.*

4.4.1 Instantiation of the Algorithm

We describe our algorithm for this problem with reference to Figure 4.1. The algorithm begins first with computing the footprint from the initial sequence $\{G_1, G_2, \ldots, G_\delta\}$. As this time we study a minimization problem (finding the smallest duration). We can directly use the the generic algorithm. In our example the walk starts from $\mathcal{G}^1[1]$ and moves a step forward because $\mathcal{G}^1[1]$ equals the footprint (by chance). This operation is repeated until $\mathcal{G}^1[3]$ that does not equal the footprint. In this case the walk goes up a step to $\mathcal{G}^2[3]$ computed from $union(\mathcal{G}^1[3], \mathcal{G}^1[4])$. After the negative equality test result on $\mathcal{G}^2[3]$, the algorithm computes $\mathcal{G}^3[3]$ from $union(\mathcal{G}^2[3], \mathcal{G}^1[5])$ which in turn does not equal the footprint. The walk moves up to $\mathcal{G}^4[3]$ that equals the footprint and continues forward as the equality test is positive. The same computations and tests are repeated until the walk traverses the hierarchy. When the walk hits the right side of the hierarchy, the algorithm returns the number b of the

current row if the super node $\mathcal{G}^b[\delta - b + 1]$ (current super node) equals the footprint. Otherwise (in the example the super node $\mathcal{G}^7[10]$ does not equal the footprint) the algorithm returns $b+1$ because of the insured equality of the super node $\mathcal{G}^{b+1}[\delta - b]$ (the last super node in the row $\mathcal{G}^{b+1}$) to the footprint ($\mathcal{G}^8[9] = G_{(9,16)}$ in the example) implied by the fact that at least one of the super nodes $\{\mathcal{G}^i[\delta - b] : 1 \leq i \leq b\}$ equals the footprint (Lemma 4.2). In the example, the algorithm returns $b = 8$. If the walk reaches the super node $\mathcal{G}^\delta[1] = G_{(1,\delta)}$, that is, the footprint (by definition), then the algorithm returns δ indicating that the bound on the footprint realization is the lifetime of $\mathcal{G}$. Intermediate super nodes (ladders) are computed the same way as the algorithm for INTERVAL-CONNECTIVITY using *union* this time (instead of *intersection*).

Lemma 4.3. *Let $\mathcal{G}^b[\delta - b + 1]$ be the last visited super node at the termination of the algorithm. If $\mathcal{G}^b[\delta - b + 1]$ equals the footprint G, then $\forall i \in [1, \delta - b]$, $G^b[i]$ (all super nodes in the row $\mathcal{G}^b$) equals the footprint G.*

Proof. According to the algorithm, the walk moves one step on the same row if the visited super node equals the footprint G and it goes up if the visited super node does not equal the footprint G. So, if $\mathcal{G}^b[\delta - b + 1]$ is the last visited super node at the termination of the algorithm and it equals the footprint G, then $\forall G^b[i] = G_{(i,j')}$ with $i \in [1, \delta - b]$, $\exists G_{(i,j)}$ that equals the footprint G (red super nodes in the walk), such that, $G_{(i,j')} = G_{(i,j)} \cup G_{(i',j')}$. From Lemma 4.2, $\forall i \in [1, \delta - b]$, $G^b[i] = G_{(i,j')}$ equals the footprint G. □

Theorem 4.2. BOUNDED-REALIZATION-OF-THE-FOOTPRINT *can be solved using $\Theta(\delta)$ elementary operations.*

Proof. From Theorem 4.1 and Lemma 4.3, the algorithm performs $\Theta(\delta)$ elementary operations (*union* and *equality test*) to solve BOUNDED-REALIZATION-OF-THE-FOOTPRINT.

4.5 Temporal Diameter

In this section, we are interested in the minimization problem of finding the *temporal diameter* of a given graph $\mathcal{G}$. We remind that, informally, the *temporal diameter* of a graph at time t (step in untimed evolving graphs) is the largest temporal distance between any pair of nodes at this time, where the temporal distance between two nodes u and v at time t in a dynamic graph is the smallest duration (number of steps in untimed evolving graphs) necessary to go from u to v starting at time t using a journey. We consider the general case of non-strict journeys, that is, an

arbitrary number of hops can be performed in the same graph of the sequence $\mathcal{G}$ (the problem in the case of strict journey can be solved in the same way as we will illustrate later on). Formally:

Definition 4.6 (Temporal distance at time t)**.** *Let $\mathcal{J}^*(u, v, \mathcal{G}, t)$ be the set of journeys from u to v whose $departure \geq t$. We define the temporal distance $tDis(u, v, \mathcal{G}, t)$ from u to v at time t by the duration $min\{arrival(\mathcal{J}(u, v)) - t : \mathcal{J}(u, v) \in \mathcal{J}^*(u, v, \mathcal{G}, t)\}$.*

Definition 4.7 (Temporal diameter at time t)**.** *The temporal diameter $tDiam(\mathcal{G}, t)$ of a dynamic graph $\mathcal{G}$ at time t is the largest temporal distance between any pair of nodes at this time: $max\{tDis(u, v, \mathcal{G}, t) : u, v \in V\}$.*

Observation 4.4. *Computing the temporal diameter of a dynamic graph $\mathcal{G}$ at time t comes to finding the smallest duration d such that the sequence $\{G_t, G_{t+1}, ..., G_{t+d-1}\}$ is temporally connected.*

Definition 4.8 (Temporal diameter of a dynamic graph $\mathcal{G}$)**.** *The temporal diameter $tmpDiam(\mathcal{G})$ of a dynamic graph $\mathcal{G}$ is the largest temporal diameter $tDiam(\mathcal{G}, t)$ at any time t: $max\{tDiam(\mathcal{G}, t) : t \in \mathcal{T}\}$.*

Observation 4.5. *Computing the temporal diameter of a dynamic graph $\mathcal{G}$ comes to finding the smallest duration d such that any sequence of length d in $\mathcal{G}$ is temporally connected.*

Definition 4.9 (TEMPORAL-DIAMETER)**.** *We will use* TEMPORAL-DIAMETER *to refer to the problem of finding $\min\{d : \forall \mathcal{G}' \subseteq \mathcal{G}$ with $|\mathcal{G}'| = d$, $\mathcal{G}'$ is temporally connected$\}$ for a given $\mathcal{G}$.*

To find the temporal diameter d we must first be able to test whether a sequence is temporally connected. It was shown in Chapter 2, that a sequence of length d is temporally connected iff its transitive closure of journeys (i.e. a directed graph where an edge (u, v) exists iff a journey from u to v exists in the corresponding sequence) is complete. Starting from the dynamic graph we can build a hierarchy of transitive closures as super nodes. Figure 4.2 shows an example of a transitive closure hierarchy. The base of the hierarchy (first row) in this case (non-strict journeys) is the sequence $\{G_1^*, G_2^*, \ldots, G_\delta^*\}$, where G_i^* is the *classical transitive closure* of G_i i.e. an edge (u, v) exists in G_i^* iff a path from u to v exists in G_i. As usual, $G_{(i,i)}$ denotes the super node $\mathcal{G}^1[i]$, and thus G_i^* in this case. Similarly, $G_{(i,j)}$, with $i < j$, corresponds to the transitive closure of the journeys within the sequence $\{G_i, G_{i+1}, ..., G_j\}$. In the example (Figure 4.2), the smallest d for which all the sequences of length d in $\mathcal{G}$ are temporally connected is $d = 4$, because their transitive

closures of journeys (super nodes of row 4) are complete graphs. So the dynamic graph $\mathcal{G}$ has temporal diameter $d = 4$.

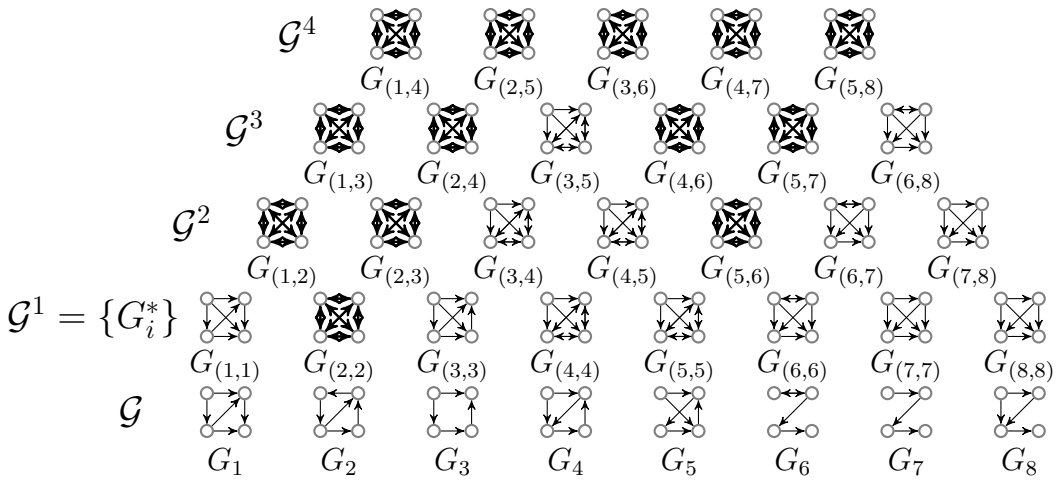


Figure 4.2: Example of a transitive closure hierarchy for a given dynamic graph $\mathcal{G}$ of length $\delta = 8$.

Observation 4.6. *The temporal diameter of a temporally connected dynamic graph $\mathcal{G}$ is the smallest d such that all transitive closure graphs in $\mathcal{G}^d$ are complete. Formally, d is the temporal diameter of a temporally connected dynamic graph $\mathcal{G}$ iff the transitive closure graph $\mathcal{G}^d[i] = G_{(i,i+d-1)}$ is complete $\forall i \in [1, \delta - d + 1]$ and $\exists \mathcal{G}^{d-1}[i'] = G'_{(i',i'+d-2)}$ with $i' \in [1, \delta - d + 2]$ such that $\mathcal{G}^{d-1}[i']$ is not complete.*

Composition operation: To solve the minimization problem TEMPORAL-DIAMETER, it is suitable to use *concatenation of transitive closures* as a composition operation (*concatenation* for short). The concatenation of transitive closures is the binary operation $cat(G_{(i,j)}, G_{(i',j')})$ that computes the transitive closure of journeys of the sequence $\{G_i, G_{i+1}, ..., G_j, G_{i'}, G_{i'+1}, ..., G_{j'}\}$ from transitive closures of journeys of the two sequences $\{G_i, G_{i+1}, ..., G_j\}$ and $\{G_{i'}, G_{i'+1}, ..., G_{j'}\}$.

A simple algorithm can compute the union graph $G_{(i,j)} \cup G_{(i',j')}$ then add an *extra-edge* (u, v) to the resulting graph if a node w exists such that $(u, w) \in E(G_{(i,j)})$ and $(w, v) \in E(G_{(i',j')})$. Figure 4.3 shows an example of transitive closures concatenation. To get concatenation result, $G_{(i,j)} \cup G_{(i',j')}$ is computed, then extra-edges are added ($G_{(i,j)\to(i',j')}$).

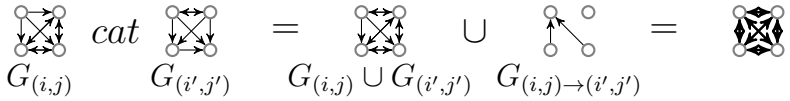


Figure 4.3: Example of transitive closures concatenation.

Observation 4.7. *The concatenation operation presented above decides whether a journey exists in a sequence* $\{G_i, G_{i+1}, ..., G_{j'}\}$ *from existing journeys in two sequences* $\{G_i, G_{i+1}, ..., G_j\}$ *and* $\{G_{i'}, G_{i'+1}, ..., G_{j'}\}$ *using the computation of extra-edges. In order for the concatenation operation to be consistent (the existence of an edge (journey) in* $G_{(i,j)}$ *and an edge in* $G_{(i',j')}$ *implies the existence of a new edge in* $G_{(i,j')}$*), the two used sequences* $\{G_i, G_{i+1}, ..., G_j\}$ *and* $\{G_{i'}, G_{i'+1}, ..., G_{j'}\}$ *must neither overlap nor be separated, we should have* $j = i' - 1$ *or* $j = i'$*. Otherwise, the computation of a transitive closure does not always allow a correct result. Figure 4.4 shows an example where the concatenation of two transitive closures of journeys* $G_{(4,8)}$ *and* $G_{(6,13)}$ *of the two sequences* $\{G_4, G_5, ..., G_8\}$ *and* $\{G_6, G_7, ..., G_{13}\}$ *does not always give a correct transitive closure of journeys* $G_{(4,13)}$ *of the sequence* $\{G_4, G_5, ..., G_{13}\}$*. Assume that only one node* w *exists such that* $(u, w) \in E(G_{(4,8)})$ *and* $(w, v) \in E(G_{(6,13)})$ *and that* (u, w) *corresponds to a journey* $u \rightsquigarrow w$ *whose* $arrival = 8$ *and* (w, v) *represents the existence of a journey* $w \rightsquigarrow v$ *with* $departure = 6$*. In this case the concatenation operation adds an edge* (u, v) *to* $G_{(4,13)}$ *even if no journey is implied by the existence of the two later ones. Actually, the concatenation operation computes in this case the transitive closure of journeys in the sequence* $\{G_4, G_5, ..., G_8, G_6, G_7, ..., G_{13}\}$*.*

Lemma 4.4. *Any computed transitive closure is correct.*

Proof. According to the *disjoint sequences property* (Lemma 4.1) guaranteed by the algorithm, in any execution, $G_{(i,j')} = cat(G_{(i,j)}, G_{(i',j')}) \Rightarrow j = i' - 1$. So, any computed transitive closure is correct. □

Test operation: To test the temporal connectivity of a sequence, it suffices to apply a *completeness test* on the transitive closure corresponding to the sequence. The completeness of a transitive closure $G_{(i,j)}$ can be tested by comparing $n(n-1)$ with $|E(G_{(i,j)})|$ that can be maintained during the construction of the transitive closure.

Observation 4.8 (Cost of the operations)**.** *Regarding the concatenation operation, the union of two transitive closures* $G_{(i,j)}$ *and* $G_{(i',j')}$ *can be computed in linear time in number of edges using an adjacency list data structure. This latter operation is dominated by the compution of extra-edges that costs* $O(|E(G_{(i',j')})| \cdot n)$*. The completeness test of a transitive closure* $G_{(i,j)}$ *can be done in constant time by checking* $|E(G_{(i,j)})|$ *maintained during the construction of the transitive closure graph.*

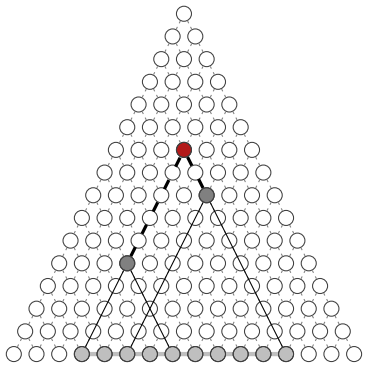


Figure 4.4: Example of a potentially incorrect compution of the transitive closure $\mathcal{G}_{(4,13)}$ from $\mathcal{G}_{(4,8)}$ and $\mathcal{G}_{(6,13)}$.

Lemma 4.5. *Let $G_{(i,j)}$ and $G_{(i',j')}$ be two super nodes and $G_{(i,j')} = cat(G_{(i,j)}, G_{(i',j')})$. $G_{(i,j)}$ is complete $\Rightarrow$ $G_{(i,j')}$ is complete.*

Proof. We have $G_{(i,j')} = cat(G_{(i,j)}, G_{(i',j')}) \Rightarrow G_{(i,j)} \subseteq G_{(i,j')}$. So, if $G_{(i,j)}$ is complete, then $G_{(i,j')}$ is complete. □

4.5.1 Instantiation of the Algorithm

The algorithm for TEMPORAL-DIAMETER is the same as for BOUNDED-REALIZATION-OF-THE-FOOTPRINT. To find the lowest row $\mathcal{G}^d$ in which all transitive closures are complete, we directly use the generic algorithm. The super nodes of the hierarchy are computed using *concatenation of transitive closures* operation. The walk is directed by the test operation that in this case is *completeness test*. It goes up only if the test is negative. When the algorithm reaches the right side of the hierarchy, it returns the number d of the current row $\mathcal{G}^d$, if the last super node is complete. Otherwise, it returns $d + 1$ ($\mathcal{G}^{d+1}[\delta - d]$, the last super node in the row $\mathcal{G}^{d+1}$, is necessarily complete because of the completeness of at least one of the transitive closures $\{\mathcal{G}^i[\delta - d] : 1 \leq i \leq d\}$, Lemma 4.5). If the last visited super node is $\mathcal{G}^\delta[1]$ and it is not complete, then the algorithm returns 0 indicating that the dynamic graph $\mathcal{G}$ is not temporally connected.

Lemma 4.6. *Let $\mathcal{G}^b[\delta - b + 1]$ be the last visited super node at the termination of the algorithm. If $\mathcal{G}^b[\delta - b + 1]$ is complete, then $\forall i \in [1, \delta - b]$, $G^b[i]$ (all super nodes in the row $\mathcal{G}^b$) is complete.*

Proof. The same argument as for Theorem 4.3 is valid. According to the algorithm, the walk moves one step on the same row if the visited super node is complete and

it goes up if the visited super node is not complete. So, if $\mathcal{G}^b[\delta - b + 1]$ is the last visited super node at the termination of the algorithm and it is complete, then $\forall \mathcal{G}^b[i] = G_{(i,j')}$ with $i \in [1, \delta - b]$, $\exists G_{(i,j)}$ that is complete (red super nodes in the walk in Figure 4.1), such that, $G_{(i,j')} = G_{(i,j)} \cup G_{(i',j')}$. From Lemma 4.5, $\forall i \in [1, \delta - b]$, $\mathcal{G}^b[i] = G_{(i,j')}$ is complete. □

Theorem 4.3. TEMPORAL-DIAMETER *can be solved using* $\Theta(\delta)$ *elementary operations.*

Proof. From Theorem 4.1 and Lemma 4.6, the algorithm performs $\Theta(\delta)$ elementary operations (*concatenation* and *completeness test*) to solve TEMPORAL-DIAMETER.

4.6 Round-trip Temporal Diameter

We will see in this section a slightly more complex case. We will look at the class *Round-trip temporal connectivity* of temporally connected graphs where there is a back-and-forth journey from each node to all other nodes (Class 6, Def. 1.18). This class characterizes an important property for distributed solutions with a termination detection or information collection algorithms. This property is defined by the fact that there is a journey $\mathcal{J}(u, v)$ from any node u in the graph to all other nodes $v \in V - u$ and there is a journey $\mathcal{J}'(v, u)$ from v to u which starts after the arrival of the journey $\mathcal{J}(u, v)$. This property of a dynamic graph $\mathcal{G}$ does not simply mean that $\mathcal{G}$ is a succession of two temporally connected sequences. It is not that simple. A back-and-forth journey from a node u to a node v ($\mathcal{J}(u, v)$, $\mathcal{J}'(v, u)$) can arrive ($arrival(\mathcal{J}'(v, u))$) even before a back-and-forth journey from a node u' to a node v' ($\mathcal{J}(u', v')$, $\mathcal{J}'(v', u')$) starts ($departure(\mathcal{J}(u', v'))$). Also the time intervals of the two back-and-forth journeys can overlap (e.g. $departure(\mathcal{J}'(u, v)) < arrival(\mathcal{J}(v', u'))$).

Let us first formally define the class in a discrete context (untimed evolving graphs).

Definition 4.10 (Round-Trip Temporal Connectivity, *Class 6*)**.** *A dynamic graph* $\mathcal{G} = \{G_1, G_2, ..., G_\delta\}$ *is* round trip temporally connected *iff for all pairs of nodes* $u, v \in V$*, there exists a back-and-forth journey, that is,* $\forall u, v \in V, \exists \mathcal{J}(u, v), \mathcal{J}'(v, u) : arrival(\mathcal{J}(u, v)) \leq departure(\mathcal{J}'(v, u))$.

What can be interesting to examine in this context using our framework is the minimization problem of finding the round trip temporal diameter rtd defined by the smallest duration in which there exist a back-and-forth journey from each node u to all other nodes $v \in V - u$.

Definition 4.11 (Round-Trip Temporal diameter)**.** *rtd is round-trip temporal diameter for a dynamic graph $\mathcal{G} = \{G_1, G_2, ..., G_\delta\}$ iff all sub-sequences $\mathcal{G}' \subseteq \mathcal{G}$ of length $|\mathcal{G}'| = rtd$ are round-trip temporally connected, that is,*
$\forall u, v \in V, t \in [1, \delta - rtd + 1], \exists \mathcal{J}(u,v), \mathcal{J}'(v,u) : t \leq departure(\mathcal{J}(u,v))$, $arrival(\mathcal{J}(u,v)) \leq departure(\mathcal{J}'(v,u))$, *and* $arrival(\mathcal{J}'(v,u)) \leq t + rtd - 1$.

Definition 4.12 (ROUND-TRIP-TEMPORAL-DIAMETER)**.** *We will use* ROUND-TRIP-TEMPORAL-DIAMETER *to refer to the problem of finding* $\min\{rtd : \forall \mathcal{G}' \subseteq \mathcal{G}$ *with* $|\mathcal{G}'| = rtd$, $\mathcal{G}'$ *is Round-Trip temporally connected*$\}$ *for a given* $\mathcal{G}$.

As ROUND-TRIP-TEMPORAL-DIAMETER is a minimization problem, the algorithm will be based on the same high-level strategy used to solve BOUNDED-REALIZATION-OF-THE-FOOTPRINT and TEMPORAL-DIAMETER. The composition and test operations must be adapted to this problem. First, let us see what test can be done on what structure to check whether a sequence is round trip temporally connected or not. In TEMPORAL-CONNECTIVITY where simple journeys must exist between every pair of nodes, we have seen that the most suitable structure for testing this condition was the transitive closure of journeys. The fact that a journey exists between two nodes adds an edge in the corresponding transitive closure. The test was then to verify if the transitive closure is a complete graph. In ROUND-TRIP-TEMPORAL-DIAMETER, back-and-forth journeys make things a little more complex.

To verify that a back-and-forth journey exists between two nodes in a given sequence of graphs $\mathcal{G}$, we need to find existing journeys between these two nodes in both directions and know their dates of departure and arrival. Then to check, it suffice to test for a pair u, v the existence of journeys $\mathcal{J}(u,v)$ and $\mathcal{J}'(v,u)$ in $\mathcal{G}$ with $arrival(\mathcal{J}(u,v)) \leq departure(\mathcal{J}'(v,u))$. So one can build a hierarchy of transitive closures, as super nodes, with dates of journeys on edges (*round trip transitive closures*) and test for each pair u, v if there is a back-and-forth journey.

Note that to test the existence of a back-and-forth journey in a given sequence, it is not required to record all dates. We can simply use the journey $\mathcal{J}(u,v)$ with the earliest arrival and the journey with the latest departure $\mathcal{J}'(v,u)$ in the sequence. This is sufficient for the test and also sufficient for the compution of a round trip transitive closure with the right dates of journeys (earliest arrival and latest departure) from two other ones.

Definition 4.13 (Round trip transitive closure)**.** *A round trip transitive closure $G_{(i,j)}$ is a transitive closure of journeys where edges represent the existence of a set of journeys ($\mathcal{J}^*(u,v,i,j)$ is not empty), that is, $(u,v) \in G_{(i,j)}$ iff at least a journey $u \rightsquigarrow v$ exists in the sequence $\{G_i, G_{i+1}, ..., G_j\}$. And edges $\{(u,v) \in E(G_{(i,j)})\}$ are labeled with two dates: $arrival(u,v,G_{(i,j)}) = min\{arrival(\mathcal{J}(u,v)) : \mathcal{J}(u,v) \in \mathcal{J}^*(u,v,i,j)\}$ and $departure(u,v,G_{(i,j)}) = max\{departure(\mathcal{J}(u,v)) : \mathcal{J}(u,v) \in \mathcal{J}^*(u,v,i,j)\}$. Labels on the same edge may or may not be departure and arrival dates of the same journey.*

Figure 4.5 shows an example of a round trip transitive closures hierarchy of a dynamic graph $\mathcal{G}$ of length $\delta = 3$. Labels arr and dep on an edge $u \xrightarrow{arr,\ dep} v$ represent respectively $arrival(u,v,G_{(i,j)})$ and $departure(u,v,G_{(i,j)})$.

The hierarchy is built using the composition operation from the first row which in this case is the sequence of classical transitive closures $\{G_{(i,i)} = G_i^*\}$ of graphs $\{G_i\}$ with dates $arrival(u,v,G_{(i,i)}) = departure(u,v,G_{(i,i)}) = i$ for all edges $\{(u,v) \in E(G_{(i,i)})\}$. Indeed, all paths in a given graph are (non-strict) journeys.

Composition operation: The composition operation in our case is the *concatenation of round trip transitive closures* with a maintenance of dates on the edges. *The concatenation of round trip transitive closures* is the binary operation $rtcat(G_{(i,j)}, G_{(i',j')})$ that computes the round trip transitive closure of journeys of the sequence $\{G_i, G_{i+1}, ..., G_j, G_{i'}, G_{i'+1}, ..., G_{j'}\}$ from round trip transitive closures of journeys of the two sequences $\{G_i, G_{i+1}, ..., G_j\}$ and $\{G_{i'}, G_{i'+1}, ..., G_{j'}\}$. The algorithm computes first the graph $G^{\cup\circlearrowleft} = G_{(i,j)} \cup^{\circlearrowleft} G_{(i',j')}$ which is the union graph $G_{(i,j)} \cup G_{(i',j')}$ with $arrival(u,v,G^{\cup\circlearrowleft}) = min(arrival(u,v,G_{(i,j)}), arrival(u,v,G_{(i',j')}))$ and $departure(u,v,,G^{\cup\circlearrowleft}) = max(departure(u,v,G_{(i,j)}), departure(u,v,G_{(i',j')})$ if $(u,v) \in G_{(i,j)} \cap G_{(i',j')}$. Otherwise, the edge is added with the initial dates.

A graph of *extra-edges* $G_{(i,j)\to(i',j')}$ is then computed as follows:
$(u,v) \in G_{(i,j)\to(i',j')}$ iff a set of nodes $extra = \{w : (u,w) \in E(G_{(i,j)})$ and $(w,v) \in E(G_{(i',j')})\}$ exists (not empty).
$arrival(u,v,G_{(i,j)\to(i',j')}) = min\{arrival(w,v,G_{(i,j)})\}$ and
$departure(u,v,G_{(i,j)\to(i',j')}) = max\{departure(u,w,G_{(i,j)})\} : w \in extra$.

Finally, the round trip transitive closure $rtcat(G_{(i,j)}, G_{(i',j')}) = G^{\cup\circlearrowleft} \cup^{\circlearrowleft} G_{(i,j)\to(i',j')}$

Figure 4.6 shows an example of round trip transitive closures concatenation $rtcat(G_{(1,5)}, G_{(6,7)}) = G_{(1,7)}$. In this example, to get $G_{(1,7)}$, $G_{(1,5)} \cup^{\circlearrowleft} G_{(6,7)}$ is computed then round trip extra-edges are added ($G_{(1,5)\to(6,7)}$) from $G_{(1,5)} \cup^{\circlearrowleft} G_{(6,7)} \cup^{\circlearrowleft}$

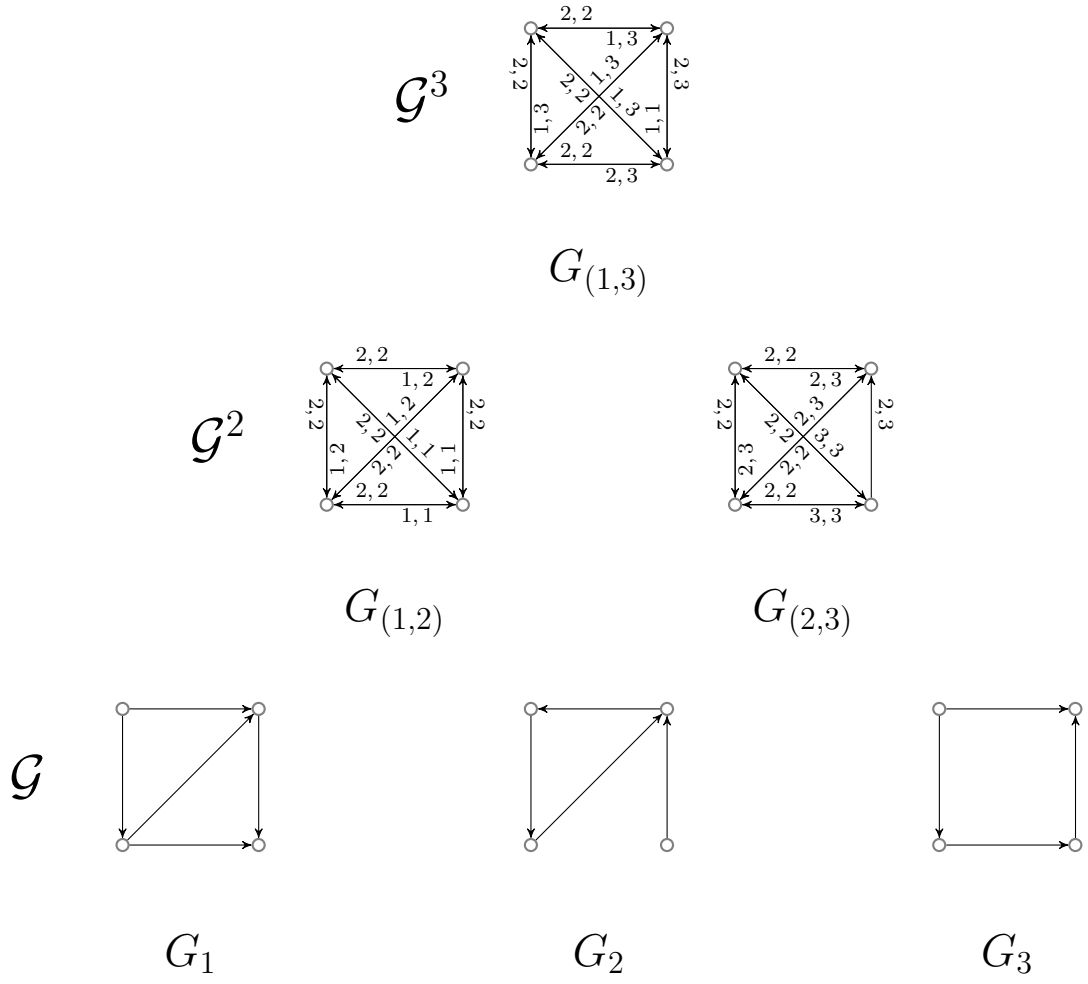


Figure 4.5: Example of a round trip transitive closure of journeys of a round trip temporally connected dynamic graph $\mathcal{G}$ of length $\delta = 3$.

$G_{(1,5)\to(6,7)}$.

Test operation: The test operation used for this problem is the *round trip completeness test* (*completeness test* for short), that is, testing if a sequence $\{G_i, G_{i+1}, ..., G_j\}$ is round trip temporally connected comes to test if its round trip transitive closure $G_{(i,j)}$ is a complete graph and for all edges $\{(u, v) \in E(G_{(i,j)})\}$, $arrival(u, v, G_{(i,j)}) \leq departure(v, u, G_{(i,j)})$. If the test is positive, $G_{(i,j)}$ is said to be round trip complete.

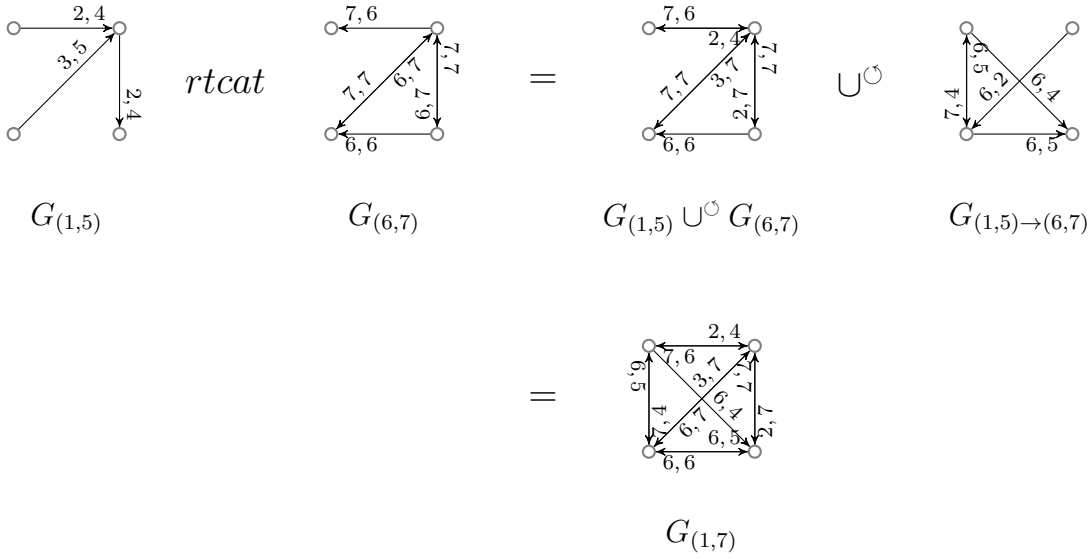


Figure 4.6: Example of round trip transitive closures concatenation.

Observation 4.9 (Cost of the operations)**.** *As the concatenation operation for* TEMPORAL-DIAMETER*, the concatenation of two round trip transitive closures $G_{(i,j)}$ and $G_{(i',j')}$ can be computed in $O(|E(G_{(i',j')})| \cdot n)$. The completeness test can be done in time linear in the number of edges by verifying the condition on the dates for each pair of edges (u, v) , (v, u).*

Lemma 4.7. *Let $G_{(i,j)}$ and $G_{(i',j')}$ be two super nodes and $G_{(i,j')} = rtcat(G_{(i,j)}, G_{(i',j')})$. $G_{(i,j)}$ is round trip complete $\Rightarrow$ $G_{(i,j')}$ is round trip complete.*

Proof. Let $G_{(i,j')} = rtcat(G_{(i,j)}, G_{(i',j')})$. According to the round trip concatenation operation described above, we have:

(a) $G_{(i,j)}$ is round trip complete $\Rightarrow$ $G_{(i,j')}$ is a complete graph.

(b) According to the disjoint sequences property (Lemma 4.1), we have:
$G_{(i,j)}$ is round trip complete $\Rightarrow \forall (u, v) \in E(G)$,
$arrival(u, v, G_{(i,j')}) = arrival(u, v, G_{(i,j)})$ and
$departure(u, v, G_{(i,j')}) \geq departure(u, v, G_{(i,j)})$.

(c) $G_{(i,j)}$ is round trip complete $\Rightarrow \forall (u, v) \in E(G)$,
$arrival(u, v, G_{(i,j)}) \leq departure(v, u, G_{(i,j)})$ (a back-and-forth journey exists from u to v).

From (a), (b), and (c), $G_{(i,j)}$ is round trip complete $\Rightarrow G_{(i,j')}$ is a complete graph and $\forall(u,v) \in E(G)$, $arrival(u,v,G_{(i,j')}) \leq departure(v,u,G_{(i,j')})$. So, $G_{(i,j)}$ is round trip complete $\Rightarrow G_{(i,j')}$ is round trip complete. □

4.6.1 Instantiation of the Algorithm

The algorithm for ROUND-TRIP-TEMPORAL-DIAMETER is the same as that for TEMPORAL-DIAMETER. It uses *concatenation of round trip transitive closures* and *round trip completeness test* as composition and test operation.

When the walk hits the right side of the hierarchy, it returns the number rtd of the current row if the completeness test on the last super node of this row is positive, otherwise it returns $rtd+1$ (same argument as for TEMPORAL-DIAMETER). If the last visited super node is $\mathcal{G}^{\delta}[1]$ and the test is negative, then 0 is returned indicating that $\mathcal{G}$ is not round trip temporally connected. Super nodes are computed the same way as in the algorithm for TEMPORAL-DIAMETER. The *disjoint sequences property* from Lemma 4.1 is always guaranteed and ensures the correctness of the concatenation of round trip transitive closures operation in the same way as for TEMPORAL-DIAMETER (the same argument as in Observation 4.7 and Lemma 4.4).

Lemma 4.8. *Let $\mathcal{G}^b[\delta-b+1]$ be the last visited super node at the termination of the algorithm. If $\mathcal{G}^b[\delta-b+1]$ is round trip complete, then $\forall i \in [1,\delta-b]$, $\mathcal{G}^b[i]$ (all super nodes in the row $\mathcal{G}^b$) is round trip complete.*

Proof. The same argument as for Theorem 4.6 is valid. According to the algorithm, the walk moves one step on the same row if the visited super node is round trip complete and it goes up if the visited super node is not round trip complete. So, if $\mathcal{G}^b[\delta-b+1]$ is the last visited super node at the termination of the algorithm and it is round trip complete, then $\forall \mathcal{G}^b[i] = G_{(i,j')}$ with $i \in [1,\delta-b]$, $\exists G_{(i,j)}$ that is round trip complete (red super nodes in the walk in Figure 4.1), such that, $G_{(i,j')} = G_{(i,j)} \cup G_{(i',j')}$. From Lemma 4.7, $\forall i \in [1,\delta-b]$, $G^b[i] = G_{(i,j')}$ is round trip complete. □

Theorem 4.4. ROUND-TRIP-TEMPORAL-DIAMETER *can be solved using $\Theta(\delta)$ elementary operations.*

Proof. From Theorem 4.1 and Lemma 4.8, the algorithm performs $\Theta(\delta)$ elementary operations (*round trip concatenation* and *round trip completeness test*) to solve ROUND-TRIP-TEMPORAL-DIAMETER.

Conclusions

In this chapter we generalized the presented framework and the algorithm for INTERVAL-CONNECTIVITY to solve other problems on dynamic graphs. We studied the minimization problems of finding the temporal diameter, the round trip temporal diameter of a given dynamic graph $\mathcal{G} = \{G_1, G_2, ..., G_\delta\}$ and the bound of its footprint realization. We proposed algorithms for these problems within the same framework.

In our study, we focused on algorithms using only two elementary operations *composition* and *test* operations. This approach is suitable for a high-level study of these problems when the details of changes between successive graphs in a sequence are arbitrary. If the evolution of the dynamic graph is constrained in some ways (e.g., bounded number of changes between graphs), then one could benefit from the use of more sophisticated data structures to lower the complexity of the problem.

A natural extension of our investigation would be a similar study for other classes and properties of dynamic graphs, as identified in Casteigts *et al.* [2012].

Chapter 5

Maintaining a Spanning Forest in Highly Dynamic Graphs

Contents

In this chapter, we propose a solution based on random walk of tokens to maintain a forest of spanning trees in a dynamic graph. The proposed algorithm is the adaptation of a coarse-grain interaction algorithm (Casteigts *et al.* [2013a]) to the synchronous message passing model. It relies on purely localized decisions, where no global information on the graph is collected. This work is an attempt to study what can still be done in a context where no property can be guaranteed on the graph.

The results presented in this chapter were published in the 18th International Conference on Principles of Distributed Systems OPODIS (Barjon *et al.* [2014a]).

5.1 Introduction

In the context of dynamic graphs where the topology changes permanently, not only changes are frequent, but in general, they even make the graph partitioned. Clearly, the usual assumption of connectivity does not hold here, although the temporal connectivity property is often available. Also, the classical view of a graph whose dynamics corresponds to *failures* is no longer suitable in these scenarios, where dynamics is the norm rather than the exception.

This induces a shift in paradigm that strongly impacts algorithms. In fact, it even impacts the problems themselves. What does it mean, for instance, to elect a leader in such a graph? Is the objective to distinguish a unique global leader, whose leadership then takes place over time and space, or is it to *maintain* a leader in each connected component, so that the decisions concerning each component are taken quickly and locally. The same remark holds for spanning trees. Should an algorithm construct a unique, global tree whose logical edges survive intermittence, or should it build and maintain a *forest* of trees that strive to cover collectively all components in each instant? Both viewpoints make sense, and so far, were little studied in distributed computing (see e.g. Awerbuch and Even [1984a]; Casteigts *et al.* [2014] for temporal trees, Awerbuch *et al.* [2008]; Casteigts *et al.* [2013a] for maintained trees).

We focus on the second interpretation (maintenance), which reflects a variety of scenarios where the expected output of the algorithm should relate to the *immediate* configuration (*e.g.* direct social networking, swarming of flying robots, vehicles platooning on the road). A particular feature of this type of algorithms is that they never terminate. More significantly, in highly dynamic graphs, they are not even expected to stabilize to an optimal state (here, a single tree per component), unless the changes stop, which never happens. This precludes, in particular, all approaches whereby the computation of a new solution requires the previous computation to

have completed.

This chapter is an attempt to understand what can still be computed (and guaranteed) when no assumptions are made on the graph dynamics: neither on the rate of change, nor on their simultaneity, nor on global connectivity. In other words, we do not assume that the graph belongs to any of presented classes. In this seemingly chaotic context, we present an algorithm that strives to maintain as few trees per components as possible, while always guaranteeing some properties.

5.1.1 Related Work

Several works have addressed the spanning tree problem in dynamic graphs, with different goals and assumptions. Burman and Kutten [2007] and Kravchik and Kutten [2013] consider a self-stabilizing approach where the legal state corresponds to having a (single) minimum spanning tree and the faults are topological changes. The strategy consists in recomputing the entire tree whenever changes occur. This general approach, sometimes called the "blast away" approach, is meaningful if stable periods of time exist, which is not assumed here.

Many spanning tree algorithms rely on random walks for their elegance and simplicity, as well as for the inherent localized paradigm they offer. In particular, approaches that involve multiple coalescing random walks allow for uniform initialization (each node starts with the same state) and topology independence (same strategy whatever the graph). Pioneering studies involving such processes include Bar-Ilan and Zernik [1989] (for the problem of election and spanning tree), Israeli and Jalfon [1990] (mutual exclusion), and Chapter 14 of Aldous and Fill [2002] (for general analysis).

The principle of using coalescing random walks to build spanning trees in mildly dynamic graphs was used by Abbas *et al.* [2006] and Baala *et al.* [2003], where tokens are annexing territories gradually by capturing each other. Regarding dynamicity, both algorithms require the nodes to know an upper bound on the cover time of the random walk, in order to regenerate a token if they are not visited during a long-enough period of time. Besides the strength of this assumption (akin to knowing the number of nodes n, or the size of components in our case), the efficiency of the timeout approach decreases dramatically with the rate of topological changes. In particular, if they are more frequent than the cover time (itself in $O(n^3)$), then the tree is constantly fragmented into "dead" pieces that lack a root, and thus a leader.

Another algorithm based on random walks is proposed by Bernard *et al.* [2013]. Here, the tree is constantly redefined as the token moves (in a way that reminds the snake game). Since the token moves only over present edges, those edges that have disappeared are naturally cleaned out of the tree as the walk proceeds. Hence, the

algorithm can tolerate failure of the tree edges. However it still suffers from detecting the disappearance of tokens using timeouts based on the cover time, which as we have seen, suits only slow dynamics.

A recent work by Awerbuch *et al.* [2008] addresses the maintenance of *minimum* spanning trees in dynamic graphs. The paper shows that a solution to the problem can be updated after a topological change using $O(n)$ messages (and same time), while the $O(m)$ messages of the "blast away" approach was thought to be optimal. (This demonstrates, incidentally, the relevance of *updating* a solution rather than recomputing it from scratch in the case of minimum spanning trees.) The algorithm has good properties for highly dynamic graphs. For instance, it considers as natural the fact that components may split or merge perpetually. Furthermore, it tolerates new topological events while an ongoing update operation is executing. In this case, update operations are enqueued and consistently executed one after the other. While this mechanism allows for an arbitrary number of topological events *at times*, it still requires that such burst of changes are only episodical and that the graph remains eventually stable for (at least) a linear amount of time in the number of nodes, in order for the update operations to complete and thus the logical tree to be consistent with physical reality.

All the aforementioned algorithms either assume that *global update* operations (e.g. wave mechanisms) can be performed contemporaneously, or at least eventually, or that some node can collect *global information* about the tree structure. As far as dynamics is concerned, this forbids arbitrary and ever going changes to occur in the graph.

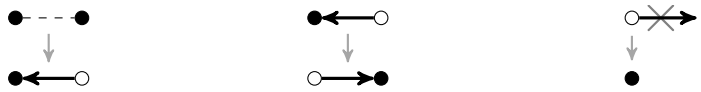
5.1.2 The Spanning Forest Principle

A purely localized scheme was proposed in Casteigts *et al.* [2013a] and Casteigts [2006] for the maintenance of a (non-minimum) spanning forest in unrestricted dynamic graphs, using a coarse grain interaction model inspired from graph relabeling systems (Litovsky *et al.* [2001]). It can be described informally as follows:

Initially every node hosts a token and is the *root* of its own individual tree. Whenever two roots arrive at the endpoints of a same edge (see merging rule on Figure 5.1), one of them destroys its tokens and selects the other as parent (*i.e.* the trees are merged). The rest of the time, each token executes a random walk within its own tree in the search for other merging opportunities (circulation rule). Tree relations are flipped accordingly. The fact that the random walk is *confined* to the underlying tree is crucial and different from all algorithms discussed above, in which they were free to roam everywhere without restriction. This simple feature induces very attractive properties for highly dynamic graphs. In particular, whenever an edge of the tree disappears, the child side of that edge knows instantly that

no token remains on its whole subtree. It can thus regenerate a token (*i.e.* become root) *instantly*, without global concertation nor further information collection. As a result, both merging and splitting of trees are managed in a purely localized fashion.

Figure 5.2 shows an example of the algorithm execution. The transition between two steps is induced by applying one or more rules on one or different edges and / or a topology change on the graph. The choice of edges here for the application of the rules is made in a way that helps to show the evolution of the execution. In the general model, the only assumption on the rules scheduler is that a node should not be involved in several rules at the same time.



(a) Merging rule (b) Circulation rule (c) Regeneration rule

Figure 5.1: Spanning forest principle (high-level representation). *Black nodes are those having a token. Black directed edges denote child-to-parent relationships. Gray vertical arrows represent transitions.*

At an abstract graph level, this very simple scheme guarantees that the graph remains covered by a spanning forest at any time, in which 1) no cycle can ever appear, 2) maximal subtrees are always directed rooted trees (with a token at the root), and 3) every node always belongs to such a tree, whatever the chaos of topological changes. On the other hand, it is not expected to reach an optimal state where a single tree covers each connected component. Even if the graph were to stabilize, convergence to the optimum (though easy to be made certain) would not be expected to occur fast. Whether this general principle could be implemented in a message passing model remained an open question.

5.1.3 Our Contribution

This chapter provides an implementation of the spanning forest principle in the synchronous message-passing model. Due to the loss of atomicity and exclusivity in the interaction, the algorithm turns out to be much more sophisticated than its original counterpart. While still reflecting the very same high-level principle, it faces new problems that require conceptual differences. In particular, the original model prevented a node from both selecting a parent and being selected as parent simultaneously, making it easier to avoid cycle creations. One of the ingredients in the new algorithm to circumvent this problem is an original technique (which we refer to as the *unique score* technique) that consists of maintaining, graph-wide, a set of `score` variables that always remain a permutation of the set of nodes IDs. This mechanism allows us to break symmetry and avoid the formation of cycles

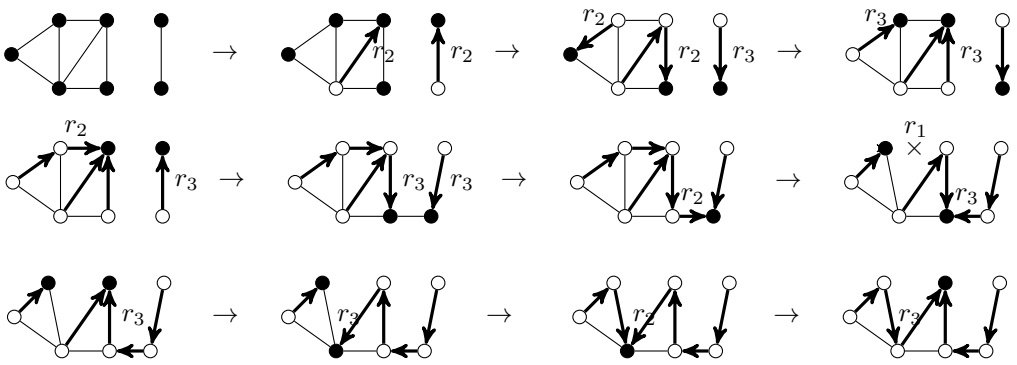


Figure 5.2: Example of the high level spanning forest algorithm execution. *Black nodes are those having a token. Black directed edges denote child-to-parent relationships. Labels on edges (right or above edges) show rules application.* r_1 = *Merging rule,* r_2 = *Circulation rule,* r_3 = *Regeneration rule.*

in a context where IDs alone could not. The paper is organized as follows. In Section 5.2, we present the model and notations that we use throughout the paper. Then Section 5.3 presents the algorithm, whose correctness analysis is outlined in Section 5.4 and detailed in Section 5.5. Section 5.6 presents experimental results that validate our algorithm.

5.2 Model and Notations

We consider in this chapter dynamic graphs that are represented as untimed evolving graph $\mathcal{G} = \{G_1, G_2, \dots\}$, such that $G_i = (V, E_i)$ (Def. 1.4). Following Kuhn *et al.* [2010], we consider a synchronous (thus rounded) computational model, where in each round i, the adversary chooses the set of edges E_i that are present. In our case, this set is arbitrary (*i.e.* the adversary is unrestricted). At the beginning of each round, each node sends a message that it has prepared at the end of the previous round. This message is sent to all its neighbors in E_i, although the list of these neighbors is not known by the node. Then it receives all messages sent by its neighbors (in the same round), and finally computes its new state and the next message. Hence, each round corresponds to three phases `(send, receive, compute)`, which corresponds to a rotation of the original model of Kuhn *et al.* [2010] where the phases are `(compute, send, receive)`. This adaptation is not necessary, but it allows us to formulate correctness of our algorithm in terms of the state of the nodes *after* each round rather than in the middle of rounds.

We assume that the nodes have a unique identifier taken from a totally ordered set, that is, for any two nodes u and v, it either holds that $ID(u) > ID(v)$ or $ID(u) < ID(v)$. A node can specify what neighbor its message is intended to

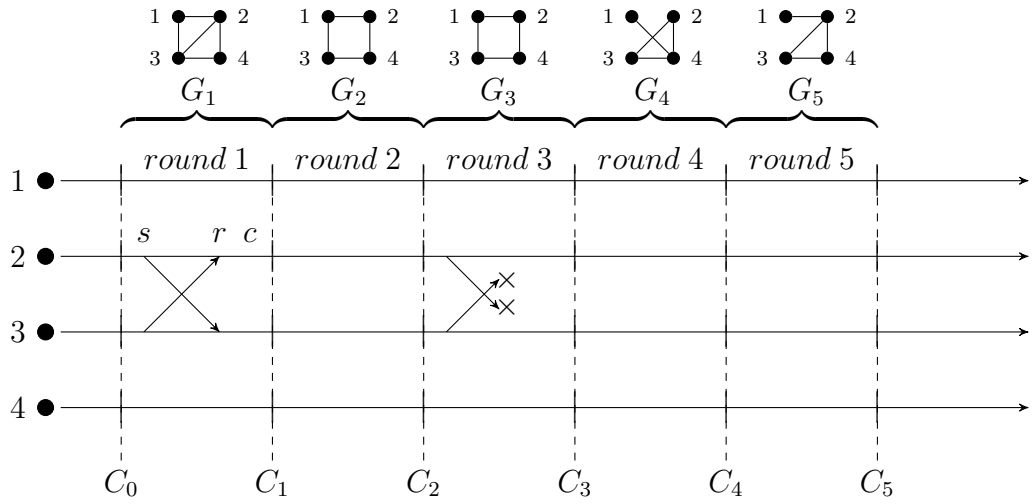


Figure 5.3: Example of the system evolution over time.

(although all neighbors will receive it) by setting the `target` field of that message. Symmetrically, the ID of the emitter of a message can be read in the `sender` field of that message. Since the edges are undirected, if u receives a message from v at round i, then v also receives a message from u at that round. And if u does not receive a message from v at round i, then v also does not receive a message from u at that round. We call this property the *reciprocity principle* and it is an important ingredient for the correctness of our algorithm.

Definition 5.1 (Configuration)**.** *A* configuration C_i *corresponds to the state of the system* after *round* i *(except for* C_0*, the initial state). Each* configuration *consists of the union of all nodes variables, defined later.*

Definition 5.2 (Execution)**.** *Using synchronous rounds allows us to represent the* execution *as a sequence of* configurations $(C_0, C_1, C_2, ..., C_i)$.

Figure 5.3 shows the evolution of the execution on a four nodes dynamic graph. The exchange of messages (steps s and r) and the local computation (step c) are illustrated in two rounds between the two nodes 2 and 3 (the same process is executed by all nodes) to show the possible cases (reception and loss of messages) and consistency with the state of the graph.

5.3 The Spanning Forest Algorithm

5.3.1 State Variables

Besides the `ID` variable, which we assume is externally initialized, each node has a set of variables that reflects its situation in the tree: `status` accounts for the possession of a token (`T` if it has a token, `N` if it does not); `parent` contains the `ID`

of this node's parent ($\perp$ if it has none); `children` contains the set of this node's children ($\emptyset$ if it has none). Observe that both variables `status` and `parent` are somewhat redundant, since in the spanning forest principle (see Section 5.1.2) the possession of a token is equivalent to being a root. Our algorithm enforces this equivalence, yet, keeping both variables separated simplifies the description of the algorithm and our ability to think of it intuitively. Variable `neighbors` contains the set of nodes from which a message was received in the last reception. These neighbors may or may not belong to the same tree as the current node. Variable `contender` contains the `ID` of a neighbor that the current node considers selecting as parent in the next round (or $\perp$ if there is no such node). Finally, the variable `score` is the main ingredient of our cycle-avoidance mechanism, whose role is described below.

Initial values

All the nodes are uniformly initialized. They are initially the root of their own individual tree (*i.e.* `status` $= T$, `parent` $= \perp$, and `children` $= \emptyset$). They know none of their neighbors (`neighbors` $= \emptyset$), have no contenders (`contender` $= \perp$), and their `score` is set to their own `ID`.

To summarize:

Variable	Description	Initial value
`ID`	Unique node identifier.	Integer in $[1, n]$
`status`	`T` if the current node has a token, `N` if it does not.	`T`
`parent`	`ID` of current node's parent.	$\perp$
`children`	Set of current node's children `ID`.	$\emptyset$
`neighbors`	Set of nodes `IDs` from which a message was received in the last reception.	$\emptyset$
`contender`	`ID` of a neighbor that the current node considers selecting as parent in the next round.	$\perp$
`score`	Unique integer used for cycle-avoidance mechanism.	`ID`

Table 5.1: List of local variables.

5.3.2 Structure of a Message (and associated variables)

Messages are composed of a number of fields: `sender` is the ID of the sending node; `senderStatus` its status (either `T` or `N`); and `score` its score when the

message was prepared. The field `action` is one of $\{FLIP, SELECT, HELLO\}$. Informally, $SELECT$ messages are sent by a root node to another root node to signify that it "adopts" it as a parent (merging operation); $FLIP$ messages are sent by a root node to circulate the token to one of its children (circulation operation); $HELLO$ messages are sent by a node by default, when none of the other messages are sent, to make its presence and status known by its neighbors. Finally, `target` is the ID of the neighbor to which a FLIP or a SELECT message are intended ($\perp$ for HELLO messages).

Received messages are stored in a variable `mailbox`, which is a map collection whose *keys* are the senders ID (*i.e.,* a message whose sender ID is u can be accessed as `mailbox[`u`]`). In each round, the algorithm makes use of a `RECEIVE()` function that clears the mailbox and fill it with all the messages received in that round (one for each physical neighbor). A node can thus update the set of its neighbors by fetching the *keys* of its mailbox. Similarly, it can eliminate from its list of children those nodes which are no more neighbor.

As mentioned above, every node prepares at the end of a round the message to be sent at the beginning of the next round. This message is stored in a variable `outMessage`. We allow the short hand `m` $\leftarrow (a, b, c, d, e)$ to define a new message m whose emitter is node a (with status b and score e); target is node d; and action is c.

Initial values

The mailbox is initially empty (`mailbox` $= \emptyset$) and `outMessage` is initialized to $(ID, T, HELLO, \perp, ID)$.

5.3.3 Informal Description of the Algorithm

The algorithm implements the general scheme presented in Section 5.1.2. In this Section we explain how each of the three core operations (*merging*, *circulation*, *regeneration*) is implemented. Then we discuss the specificities of the merging operation in more detail and the problems that arise due to its entanglement with the circulation operation, a fact due to the loss of atomicity in the message passing model. The resulting solution is substantially more sophisticated than its original scheme, and yet it faithfully reflects the same high-level principle. Let us start with some generalities. In each round, each node broadcasts to its neighbors a message containing, among others, its status (`T` or `N`) and an action (SELECT, FLIP, or HELLO). Whether or not the message is intended to a specific $target$ (which is the case for SELECT and FLIP messages), all the nodes who receive it can possibly use this information for their own decisions. More generally, based on the received information and the local state, each node computes at the end of the round its

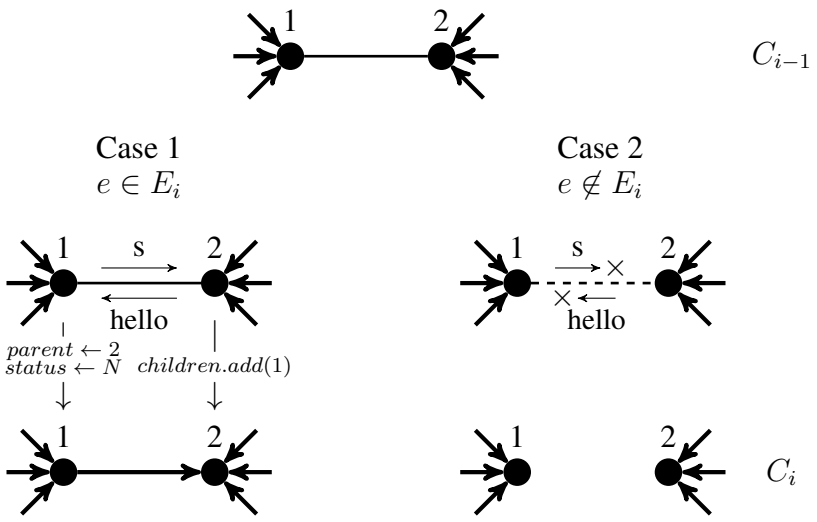


Figure 5.4: Example of local merging operation during round i in the two possible cases: $e \in E_i$ and $e \notin E_i$. We suppose that `ID=score` on all nodes.

new status and the local structure of its tree (variables `children` and `parent`), then it prepares the next message to be sent. We now describe the three operations. Throughout the explanations, the reader is invited to refer to Figure 5.7, where an example of execution involving all of them is shown. All details are also given in the listings of Algorithm 6 and 7.

Merging

If a root (*i.e.* a node having a token), say v, detects the existence of a neighbor root with higher `score` than its own, then it considers that node as a possible `contender`, *i.e.* as a node that it might select as a parent in the next round. If several such roots exist, then the one with highest score, say u, is chosen. At the beginning of the next round, v sends a $SELECT$ message to u to inform it that it is its new parent. Two cases are possible: either the considered edge is still present in that round, or it disappeared in-between both rounds. If it is still present, then u receives the message and adds v to its children list, among others (Line 16). As for v, it sets its `parent` variable to u and its `status` to N (Lines 8 and 9). If the edge disappeared, then u does not receive the message, which is lost. However, due to the reciprocity of message exchange, v does not receive a message from u either and thus simply does not executes the corresponding changes. By the end of the round, either the trees are properly merged, or they are properly separated. Figure 5.4 shows the merging at round i of two trees with roots v with score 1 and u with score 2. The figure shows the local operation in the two possible cases: $(u, v) \in E_i$ and $(u, v) \notin E_i$.

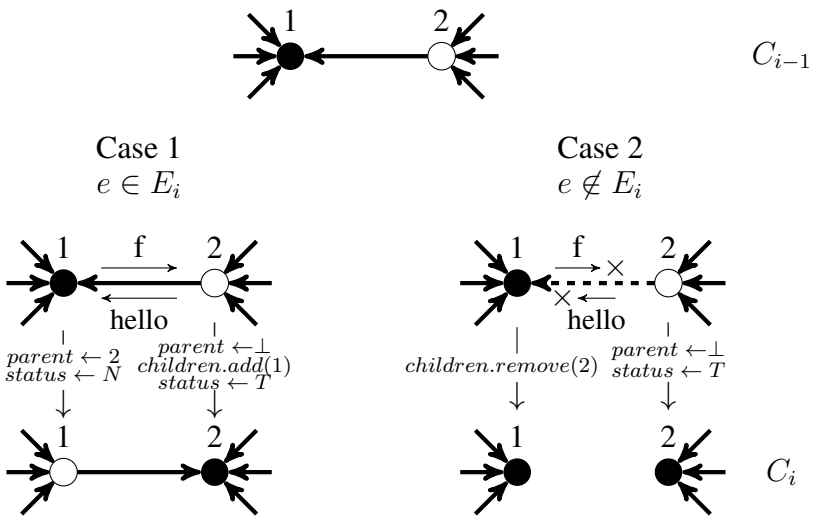


Figure 5.5: Example of local token circulation operation from node 1 to node 2 during round i in the two possible cases: $e \in E_i$ and $e \notin E_i$. The possession of the token by a node is represented by its black filling color. We suppose that `ID=score` on all nodes.

Circulation

If a root v does not detect another root with higher score, then it selects one of its children at random, if it has any (see Line 27), otherwise it simply remains root. Randomness is not a strict requirement of our algorithm and replacing it with any deterministic strategy would not affect correctness of the algorithm. Once the child is chosen, say u, the root prepares a FLIP message intended to u, and sends it at the beginning of the next round. Two cases are again possible, whether or not the edge (u, v) is still present in that round. If it is still present, then u receives the message, it updates its status and adds v to its children list, among others (Lines 15 and Line 16). As for v, it sets its `parent` variable to u and its `status` to N (Lines 8 and 9). If the edge disappeared, then v can detect it as before simply does not executes the corresponding changes. Node u, on the other hand, detects that the edge leading to its current parent disappeared, thus it regenerates a token (discussed next). Notice that in the absence of a merging opportunity, a node receiving the token in round i will immediately prepare a FLIP message to circulate the token in the next round. Unless the tree is composed of a single node, the tokens are thus moved in each round. In order for them to remain detectable in this case, the status announced in $FLIP$ messages is `T` (whereas it is `N` for $SELECT$ messages). Figure 5.5 shows the token circulation operation from node v with the score 1 to node u with the score 2 during round i in the two cases: $(u, v) \in E_i$ and $(u, v) \notin E_i$.

Regeneration

The first thing a non-root node does after receiving the messages of the current round is to check whether the edge leading to its current parent is still present. If the edge disappeared, then the node regenerates a root directly (Line 7). A nice property of the spanning forest principle is that this cannot happen twice in the same tree. And if a tree is broken into several pieces simultaneously, then each of the resulting subtree will have exactly one node performing this operation. Figure 5.6 shows the regeneration local operation by the node v with the score 2 in round i.

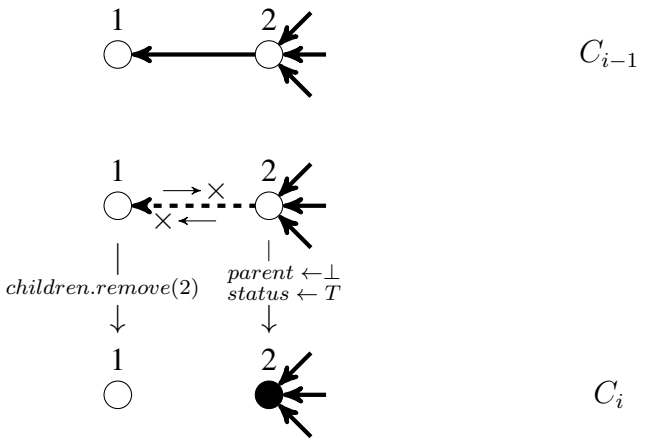
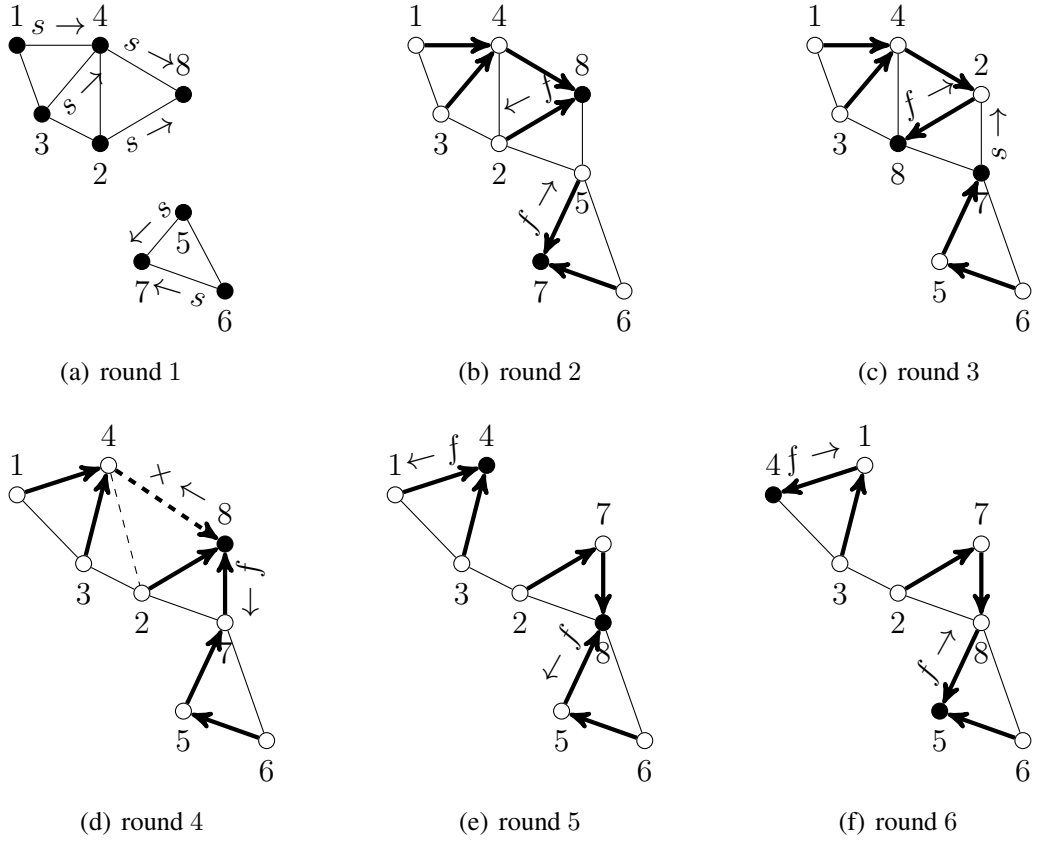


Figure 5.6: Example of local token regeneration operation by a node 2 during round i. The possession of the token by a node is represented by its black filling color. We suppose that `ID=score` on all nodes.

The unique score technique

Unlike the high-level graph model from Casteigts *et al.* [2013a], in which the merging operation involved two nodes in an *exclusive* way, the non-atomic nature of message passing allows for a *chain* of selection that may involve an arbitrary long sequence of nodes (e.g. a selects b, b selects c, and so on). This has both advantages and drawbacks. On the good side, it makes the initial merging process very fast (see rounds 1 and 2 in Figure 5.7 to get an example). On the bad side, it is the reason why scores need to be introduced to avoid cycles. Indeed, relying only on a mere comparison of `ID` to avoid cycles is not sufficient. Consider a chain of selection in round i that ends up at some root node u. Nothing prevents u to have passed the token to a lower-ID child, say v, in the previous round $i-1$ (that same round when u's status T was overheard by the next-to-last root in the chain). Now,



(a) round 1 (b) round 2 (c) round 3

(d) round 4 (e) round 5 (f) round 6

Figure 5.7: Example of execution of the algorithm which illustrates all types of operations: parent selection ($s \rightarrow$), token circulation ($f \rightarrow$), and tree disconnection ($\times \leftarrow$). *The first two symbols represent FLIP or SELECT messages to be sent in the next round. Black (resp. white) nodes are those (not) having a token at the beginning of the round. Tree edges are represented by bold directed edges. Dash edges have just disappeared.*

nothing again prevents v to have selected one of the nodes in the selection chain in round i, thereby creating a cycle. The score mechanism prevents such a situation by enforcing that after each FLIP, the new root has a larger score than its predecessor (see Lines 9 and 13 in Algorithm 7). The score mechanism also guarantees that the current set of scores (graph-wide) is always a permutation of the initial set of scores. Hence, scores are always unique. All of these elements are crucial ingredients in the proofs of correctness of Section 5.4. Figure 5.8 shows an example of cycle formation from a configuration C_{i-1} in the case where the score technique is not used (a). The figure shows in the second case the execution of the algorithm on the same sequence of graphs and from the same configuration C_{i-1} using the score technique allowing the cycle formation avoidance.

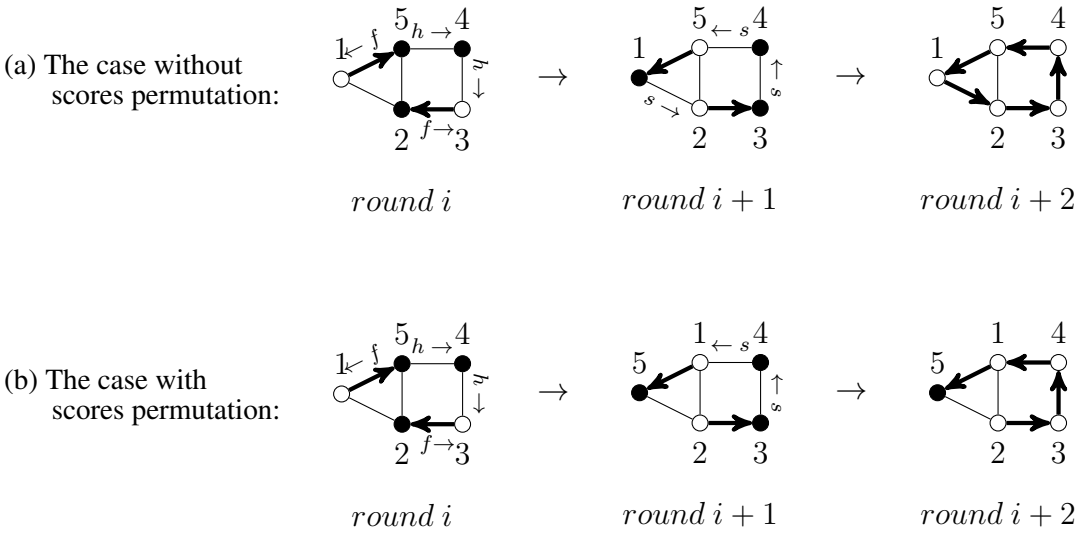


Figure 5.8: (a) Example of cycle formation in the case where the unique score technique is not used. (b) Example of cycle formation avoidance using the unique score technique.

A note about convergence

Each token performs a random walk in its underlying tree. Hence, unless some of the trees are bipartite (Figure 5.9), the configuration will eventually (and with high probability) stabilize into a single tree per connected component if the graph stops changing. Although convergence is not the main focus here, we believe that pathetic scenarios (Figure 5.9) where some trees are bipartite can easily be avoided, by making the tokens stop for a random additional round at the nodes (*lazy* walk). This way, the symmetry of bipartiteness is eventually broken *w.h.p.*

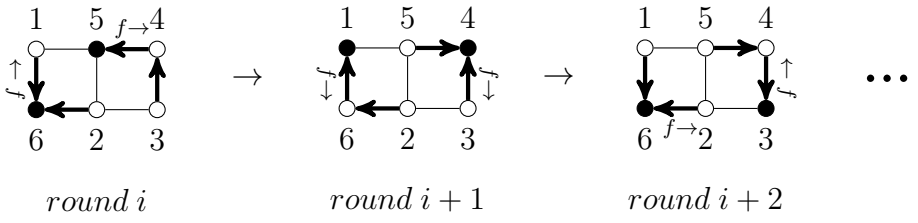


Figure 5.9: Example of adjacent trees insulation.

```
1  repeat
2      SEND(outMessage);
3      mailbox ← RECEIVE();          // Received messages,
         indexed by sender ID
4      neighbors ← mailbox.keys();   // All the senders
         IDs
5      children ← children ∩ neighbors
       // Regenerates a token if parent link is lost
6      if status=N ∧ parent ∉ neighbors then
7          BECOME_ROOT();
       // Checks if the outgoing FLIP or SELECT (if any)
         was successful
8      if outMessage.action ∈ {FLIP,SELECT} ∧
       outMessage.target ∈ neighbors then
9          ADOPT_PARENT(outMessage)
       // Processes the received messages
10     contender ← ⊥;
11     contenderScore ← 0;
12     forall message ∈ mailbox do
13         if message.target = ID then
14             if message.action = FLIP then
15                 BECOME_ROOT();
16             ADOPT_CHILD(message);     // called for both
                 FLIP or SELECT
17         else
18             if message.status = T ∧ message.score
               > contenderScore then
19                 contender ← message.ID;
20                 contenderScore ← message.score;
       // Prepares the message to be sent
21     outMessage ← ⊥
22     if status = T then
23         if contenderScore > score then
24             PREPARE_MESSAGE(SELECT, contender);
25         else
26             if children ≠ ∅ then
27                 PREPARE_MESSAGE(FLIP, random(children));
28     if outMessage = ⊥ then
29         PREPARE_MESSAGE(HELLO, ⊥);
30 ;
```

Algorithm 6: Main Algorithm

```
1  procedure BECOME_ROOT
2  |  status ← T;
3  |  parent ← ⊥;

4  procedure ADOPT_PARENT(outMessage)
5  |  status ← N;
6  |  parent ← outMessage.target;
7  |  if outMessage.action = FLIP then
8  |  |  children ← children\parent;
9  |  |  score ← min(score, mailbox[parent].score);

10 procedure ADOPT_CHILD(message)
11 |  children.add(message.ID);
12 |  if message.action = FLIP then
13 |  |  score ← max(score, message.score);

14 procedure PREPARE_MESSAGE(action, target)
15 |  switch action do
16 |  |  case SELECT
17 |  |  |  outMessage ← (ID, N, SELECT, target,
   |  |  |  score);
18 |  |  case FLIP
19 |  |  |  outMessage ← (ID, T, FLIP, target, score);
20 |  |  case HELLO
21 |  |  |  outMessage ← (ID, status, ⊥, ⊥, score);
```

Algorithm 7: Functions called in Algorithm 6.

5.4 Outline of the Correctness Analysis

This section summarizes the correctness analysis of our algorithm, whose detail (proofs of all lemmas and theorems) can be found in Section 5.5. We first define a handful of instrumental concepts that help minimize the number of properties to be proven. Then, as we start formulating the key properties to be proved, we adopt concise notations regarding the state of the system. Precisely, we denote by $(i^-)u.varname$ (resp. $(i^+)u.varname$) the value of variable $varname$ at node u before (resp. after) round i. Notice that for any node u, round i, and variable $varname$, we have $(i^+)u.varname = ((i+1)^-)u.varname$. We use whichever notation is the most convenient in the given context.

5.4.1 Helping Definitions

These definitions are not specific to our algorithm, they are general graph concepts that simplify the subsequent proofs.

Definition 5.3 (Pseudotree and pseudoforest)**.** *A directed graph whose vertices have outdegree at most 1 is a* pseudoforest*. A vertex whose outdegree is 0 is called a* root*. The weakly connected components of a pseudoforest are called* pseudotrees*.*

Lemma 5.1. *A pseudotree has at most one root.*

Proof. By definition, a pseudotree $\mathcal{T} = (V_\mathcal{T}, E_\mathcal{T})$ is connected, thus $|E_\mathcal{T}| \geq |V_\mathcal{T}| - 1$. If $\mathcal{T}$ has several roots, then at least two nodes in $V_\mathcal{T}$ have no outgoing edge. Since the others have at most one, we must have $|E_\mathcal{T}| \leq |V_\mathcal{T}| - 2$, which is a contradiction. □

Lemma 5.2. *If a pseudotree $\mathcal{T}$ contains a root r, then it has no cycle.*

Proof. Let $V_1 \subset \mathcal{T}$ be the set of nodes at distance 1 from $V_0 = \{r\}$. Since r has outdegree 0, there is an edge from each node in V_1 to r. Since $\mathcal{T}$ is a pseudotree, these nodes have no other outgoing edge than those ending up in V_0. The same argument can be applied inductively, all nodes at distance i having no other outgoing edges than those ending up in V_{i-1}. □

Definition 5.4 (Correct tree and correct forest)**.** *At the light of Lemma 5.1 and 5.2, we define a* correct tree *(or simply a* tree*) as a pseudotree in which a root can be found. We naturally define a* correct forest *(or simply a* forest*) as a pseudoforest whose pseudotrees are trees.*

Finally, because forests are considered in a spanning context, we say that a pseudoforest $\mathcal{F}$ is a correct forest *on graph* G iff $\mathcal{F}$ is a correct forest *and* $\mathcal{F}$ is a subgraph of G. Defining correct trees as pseudotrees in which a root can be found is the key. When the moment arrives, this will allow us to reduce the correctness of our algorithm to the presence of a root in each pseudotree.

5.4.2 Consistency

Forest consistency

At the end of a round, the state of an edge (whether it belongs to a tree, and if so, in what direction) must be consistently decided at both endpoints:

Definition 5.5 (forest consistency)**.** *The configuration C_i is forest consistent if and only if for all nodes u, $(i^+)u.parent = v \Leftrightarrow u \in (i^+)v.children$.*

The proof of forest consistency is inductively established by Theorem 5.1, based on consistency of the initial configuration (Lemma 5.3) and the maintenance the consistency over the rounds (Lemma 5.18). Forest consistency allows us to reduce the output of interest of the algorithm after each round i to the mere `parent` variable.

Graph consistency

At the end of round i, the values of all `parent` variables should be consistent with the underlying graph G_i.

Definition 5.6 (graph consistency)**.** *The configuration C_i is graph consistent if and only if for all nodes u, $(i^+)u.parent = v \Rightarrow (u, v) \in E_i$.*

This property is established by Corollary 5.1. Graph consistency allows us to say that the output of the algorithm forms a pseudoforest on G_i.

Definition 5.7 (Resulting forest)**.** *Given a round $i \geq 1$, occurring on graph G_i, the directed graph $\mathcal{F}_i = (V, E_{\mathcal{F}_i})$ such that $E_{\mathcal{F}_i} = \{(u, v) : (u, v) \in E_i, (i^+)u.parent = v\}$ is called the* pseudoforest *resulting from round i.*

State consistency

As explained in Section 5.3.1, the variables `parent` and `status` are somewhat redundant, since the possession of a token is synonymous with being a root. The equivalence between both variables after each round is established in Lemma 5.4. The main advantage of this equivalence is that it allows us to formulate and prove a large number of lemmas based on whichever of the two variables is the most convenient (and intuitive) for the considered property.

5.4.3 Correctness of the Forest

In this section, we prove that the resulting forest is always correct (Definition 5.4). To achieve that goal, we first define a validity criterion at the node level, which recursively ensures the correctness of the pseudotree this node belongs to thanks to Definition 5.4 (*i.e.* the existence of a root implies correctness).

Definition 5.8. *A node u is said to be valid at the beginning of round i if either $(i^-)u.status = T$ or $(i^-)u.parent$ is valid.*

The correctness of the whole forest can thus be established through showing that, first, it is initially correct (Lemma 5.3) and, second, if it is correct after round i, then it is correct after round $i+1$ (Theorem 5.2). The latter is difficult to prove, and it involves a number of intermediate steps that correspond to a case analysis based on every action a node can perform (sending FLIP messages, SELECT messages, etc.).

We first prove that a node u that sends a successful FLIP to v in a round, is valid at the end of that round (lemma 5.23) because at the end of that round v is a root. The proof relies on the fact that during a given round, a node cannot receive a FLIP and send a SELECT or a FLIP (lemma 5.20).

We then prove some necessary properties on the `score` variable at each node. For instance, a node changes its score at most once during a round (Lemma 5.25

and 5.26). Also, the set of all scores are a permutation of the node identifiers after each round (Lemma 5.27).

Then we prove that a node that sends a successful SELECT in a round i, is valid at the end of that round (Lemma 5.36). This part is the most technical and is the one that proves that chains of selection can not create cycles thanks to the property that score variables remain a permutation of all nodes IDs.

Finally, we prove that all roots at the beginning of a round are still valid at the end of the round (lemma 5.37). Therefore, if all nodes are valid at the beginning of round, then they are also valid at the end of the round (theorem 5.2). Since they are initially valid (Lemma 5.3), we conclude by induction on the number of rounds.

5.5 Detailed Proofs

5.5.1 Consistency

Lemma 5.3. *The configuration C_0 is forest consistent and graph consistent. In C_0, the resulting pseudoforest is correct.*

Proof. The `parent` variable is initialized to $\perp$. So, the configuration C_0 is forest consistent and graph consistent. Any node u belonging to the pseudotree $\mathcal{T}_u = (\{u\}, \emptyset)$. Each of these pseudotrees contains a root (u itself) and is therefore a correct tree. □

We say that u *sends a FLIP* (resp. *SELECT*) in round i if and only if $(i^-)u.outMessage.action = FLIP$ (resp. *SELECT*). We say that it sends it *to node* v if and only if $(i^-)u.outMessage.target = v$. Finally the FLIP or SELECT is said to be *successful* (resp. *failed*) if $(u, v) \in E_i$ (resp. $(u, v) \notin E_i$).

Lemma 5.4 (state consistency)**.** *For all round $i \geq 0$, and for all node u, $(i^+)u.status = T \Leftrightarrow (i^+)u.parent = \perp$*

Proof. Initially, at any node u, $u.status = T$ and $u.parent = \perp$. The change of $u.status$ to N always comes with the assignment of a non-null $identifier$ ($outMessage.target$) to $u.parent$ (procedure `ADOPT_PARENT()`), and assigning the value T to $u.status$ is always followed by the change of $u.parent$ to $\perp$ (procedure `BECOME_ROOT()`). So at any configuration, $v.parent = \perp$ if and only if $v.status = T$. □

Lemma 5.5. *If u does not send a FLIP or SELECT in round i, then u does not execute the procedure `ADOPT_PARENT()` during round i.*

Proof. The execution of the procedure `ADOPT_PARENT()` by u is conditioned by the sending of a SELECT or a FLIP by u during the current round (line 8). □

Observation 5.1. *At time where a node u prepares its message to be sent during the round i, we have $u.parent = ((i-1)^+)u.parent$ (resp. $children, status$).*

Lemma 5.6. *If u sends a FLIP or SELECT in round i, then $(i^-)u.status = T$.*

Proof. u sends in round i the message prepared in round $i-1$. If u sends a FLIP or a SELECT in round i then in round $i-1$ `PREPARE_MESSAGE()` is called with FLIP or SELECT as action (lines 24 or 27). Both instructions are conditioned by $status = T$. □

Lemma 5.7. *If v sends a message containing T in round i, then $(i^-)v.status = T$.*

Proof.
The procedure `PREPARE_MESSAGE()` is executed by a node u in round $i-1$ to construct the message m to be sent in round i. In all cases `PREPARE_MESSAGE()` sets $m.senderStatus$ to T only if $u.status = T$.

□

Lemma 5.8. *If u sends a SELECT to v in round i, then $(i^-)u.score < ((i-1)^-)v.score$.*

Proof. The value of the $score$ field in the message sent by a node v in round $i-1$ is $((i-1)^-)v.score$.

Assumes that the node u sends a SELECT to v in a round i. So, during the round $i-1$, u sets its $contender$ variable to v and its $contenderScore$ variable to $message.score$ $message$ being the message sent by v at the begining of round $i-1$. From that time to the end of round $i-1$, $u.score$ is not modified.

So $(i^-)u.score < ((i-1)^-)v.score$, if u sends a SELECT to v in a round i. □

Lemma 5.9. *If at the beginning of round i, the configuration is forest consistent then only $(i^-)u.parent$ can send a FLIP at destination of u during the round i.*

Proof. A node v can prepare a FLIP message to the node u at then end of round $i-1$ only if $u \in (i^-)v.children$. We have $(i^-)u.parent = v$ according to the hypothesis (forest consistency at the beginning of round). Therefore, only the node $(i^-)u.parent$ can prepare a FLIP message at destination of u, at the end of round $i-1$. □

Graph consistency

Lemma 5.10. *Let u be a node such that $(i^-)u.parent \neq v \wedge (i^+)u.parent = v$. Then u sends a successful FLIP or SELECT to v during the round i.*

Proof.
The only change of $parent$ by u to a non-null identifier v in a round i is at the execution of the procedure `ADOPT_PARENT()` which is conditioned by the reception of a message from v (line 9). If u receives the message of v during round i then v effectively receives the message sent by v (*reciprocal reception property*).
□

Lemma 5.11. *Let u be a node such that $(i^-)u.parent = v \wedge (i^+)u.parent = v$. We have $(u, v) \in E_i$.*

Proof.
By Lemma 5.4, we have $(i^-)u.status = N$. So, u does not send a FLIP or SELECT during the round i (Lemma 5.6). Then, u does not execute `ADOPT_PARENT()` during the round i according to Lemma 5.5. Since $(i^+)u.parent = v$ we conclude that u does not execute the procedure `BECOME_ROOT()` during the round i. So u did receive a message from $(i^-)u.parent$ in round i. We have $(u, v) \in E_i$.
□

Corollary 5.1 (graph consistency)**.** *Every configuration is graph consistent.*

Proof. The configuration reached after any round is graph consistent (Lemmas 5.10 and 5.11). □

Forest consistency

Lemma 5.12. *If $(i^-)u.parent = v$ then $(i^+)u.parent = v$ or $(i^+)u.parent = \perp$.*

Proof. According to Lemma 5.4, we have $(i^-)u.status = N$, so u cannot send a FLIP or a SELECT in round i (by Lemma 5.6). Therefore, u does not execute `ADOPT_PARENT()` in round i (Lemma 5.5). We conclude that $(i^+)u.parent = v$ or $(i^+)u.parent = \perp$. □

Lemma 5.13. *Assume that at the beginning of round i, the configuration is forest consistent. If u receives a FLIP in round i, then it does not send a FLIP nor a SELECT in round i.*

Proof. We will establish the contraposition of the lemma statement: if u sends a FLIP or a SELECT in round i, then it does not receive a FLIP in round i. By Lemma 5.6, we have $(i^-)u.status = T$. According to Lemma 5.4, $(i^-)u.parent = \perp$. Thus according to the hypothesis (forest consistency at the beginning of round), for any node v, $u \notin (i^-)v.children$. Therefore no node has prepared a FLIP message at destination of u, in round $i - 1$. So u cannot receive a FLIP in round i. □

Lemma 5.14. *Assume that at the beginning of round i, the configuration is forest consistent. If in round i, u changes $u.parent$ to v then $u \in (i^+)v.children$: $(i^-)u.parent \neq v \wedge (i^+)u.parent = v \Rightarrow u \in (i^+)v.children$.*

Proof. u sets $u.parent$ to v only if the FLIP or SELECT was successful (Lemma 5.10). Therefore v has received the FLIP or SELECT message sent by u.

The addition of a node u to $v.children$ by v is done during the excution of the procedure `ADOPT_CHILD()` which is conditioned by the reception of a FLIP or a SELECT message m_u from u ($m_u.target = v$, line 16). The procedure `ADOPT_CHILD()` is executed after line 5 which is the only instruction that could remove u from $v.children$. So, $u \in (i^+)v.children$. We have $(i^-)u.parent \neq v \wedge (i^+)u.parent = v \Rightarrow u \in (i^+)v.children$. □

Lemma 5.15. *Assume that at the beginning of round i, the configuration is forest consistent. If in round i, v adds u to $v.children$ then $(i^+)u.parent = v$: $u \notin (i^-)v.children \wedge u \in (i^+)v.children \Rightarrow (i^+)u.parent = v$.*

Proof. v adds u to $v.children$ only if it excutes the procedure `ADOPT_CHILD()` which is conditioned by the reception of a FLIP or a SELECT sent by u. As the reception of messages is reciprocal, u also receives in round i a message from v. This satisfies the condition for u to execute the procedure `ADOPT_PARENT()` which sets $u.parent$ to v.

Only the execution of `BECOME_ROOT()` (at line 15) could modify the value of $u.parent$. This procedure would be executed only if u has received a FLIP during round i which cannot be the case. Notice that u does not receive a FLIP during the round i (Lemma 5.13). □

Lemma 5.16. *Assume that at the beginning of round i, the configuration is forest consistent. If in round i, u changes $u.parent$ from v to another value then $u \notin (i^+)v.children$: $(i^-)u.parent = v \wedge (i^+)u.parent \neq v \Rightarrow u \notin (i^+)v.children$.*

Proof.
If u changes $(i^+)u.parent$ then we have $(i^+)u.parent = \perp$ (Lemma 5.12). Only the execution of `BECOME_ROOT()` by u sets $u.parent$ to $\perp$. The procedure `BECOME_ROOT()` is executed in two cases: at the detection of a disconnection (line 7), and at the reception of a FLIP message (line 15).

In the first case, the *reciprocal reception property* ensures that v does not receive the message sent by u. So, v removes u from $children$ (line 5).

In the second case, u receives a FLIP from $(i^-)u.parent$ (Lemma 5.9). According to the *reciprocal reception property*, v receives the message sent by u during the round i. So, v executes `ADOPT_PARENT(`$(i^-)v.outMessage$`)` which removes u (i.e. $(i^-)v.outMessage.target$) from $v.children$ (line 9).

□

Lemma 5.17. *Assume that at the beginning of round i, the configuration is forest consistent. If in round i, v removes u from $v.children$ then $(i^+)u.parent \neq v$: $u \in (i^-)v.children \wedge u \notin (i^+)v.children \Rightarrow (i^+)u.parent \neq v$.*

Proof.
v removes u from $v.children$ in two cases: at the detection of a disconnection (v does not receive a message from u, line 5), and when v executes (`ADOPT_PARENT(`$(i^-).v.outMessage$`)`, line 9)

In the first case, the *reciprocal reception property* ensures that u does not receive the message sent by v during the round i. So, u becomes a root: it executes the procedure `BECOME_ROOT()` (line 7).

In the second case, v executes `ADOPT_PARENT(`$(i^-).v.outMessage$`)`. So v did send a successful FLIP or SELECT (Lemma 5.5). As v removes u from $v.children$ during the execution of `ADOPT_PARENT(`$(i^-).v.outMessage$`)`, we have $(i^-).v.outMessage.target = u$ and $(i^-).v.outMessage.action = FLIP$ (see the procedure `ADOPT_PARENT(`$outMessage$`)`). So v sends a successful FLIP to u during round i. Therefore, in round i, u executes the procedure `BECOME_ROOT()` (line 15): u sets $u.parent$ to $\perp$.

□

Lemma 5.18 (Forest Consistency)**.** *Let i be a round starting from a forest consistent configuration. The configuration reached at the end of round i is forest consistent*

Proof. The configuration after the round i is forest consistent according to Lemmas 5.14, 5.15, 5.16, 5.17. Notice that in the case where u does not change the value of its parent variable (*resp.* u stays in $v.children$) during round i, at the end of round i the forest consistency property is preserved according to the contraposition of Lemma 5.17 (*resp.* contraposition of Lemma 5.16) and the hypothesis. □

Theorem 5.1 (Consistency)**.** *Every configuration is forest consistent.*

Proof. C_0 is forest consistent (Lemma 5.3). The configuration reached after any round is forest consistent (Lemma 5.18). □

5.5.2 Correctness of the Forest

Correctness of the resulting forest after token circulation

Lemma 5.19. *Let v be a node. Only $(i^-)v.parent$ can send a FLIP at destination of v during the round i.*

Proof. At the beginning of round i, the configuration is forest consistent (Theorem 5.1). Therefore, only the node $(i^-)v.parent$ can prepare a FLIP message at destination of v, at the end of round $i-1$ (Lemma 5.9). □

Lemma 5.20. *If u receives a FLIP in round i, then it does not send a FLIP nor a SELECT in round i.*

Proof. At the beginning of round i, the configuration is forest consistent (Theorem 5.1). Therefore no node has prepared a FLIP message at destination of u, in round $i-1$ (Lemma 5.13). □

Lemma 5.21 (Adoption)**.** *If u sends a successful FLIP or SELECT to v in round i, then $(i^+)u.status = N$ and $(i^+)u.parent = v$.*

Proof.
In round i, $u.outMessage.action = FLIP$ or $SELECT$ and $v \in (i^+)u.neighbors$. During the round i, u executes the procedure `ADOPT_PARENT()` (line 9) which sets $(i^+)u.parent$ to v. According to Lemma 5.20, u did not receive any FLIP message during the round i. Only an execution of `BECOME_ROOT()` by u at line 15 can change the value of $u.parent$ during the round i. This line is not executed during round i.

□

Lemma 5.22. *If u sends a successful FLIP to v, then $(i^+)v.status = T$.*

Proof.
v received mu in round i, so $(u, v) \in E_i$. v executes the procedure `BECOME_ROOT()` that changes $v.status$ to T. After the execution of line 9, no instruction can set $v.status$ to N until the end of round i. So $(i^+)v.status = T$.

□

Lemma 5.23. *If u sends a successful FLIP in round i, then u is valid after round i.*

Proof. By Lemmas 5.21 and 5.22 u's parent has a status T after round i. □

Proofs on score permutations

Lemma 5.24. *If u sends a successful FLIP to v, then $(i^-)u.score \leq (i^+)v.score$.*

Proof. u sent a message mu to v at the beginning of round i such that $mu.action =$ FLIP, $mu.target = v.ID$ and $mu.score = (i^-)u.score$. v received mu in round i, so $(u, v) \in E_i$. v executes the procedure `ADOPT_CHILD(`mu`)` at line 16 in round i. This procedure sets the current score of v to $max(v.score, mu.score)$, as $mu.score = (i^-)u.score$. After the execution of this instruction, we have $mu.score = (i^-)u.score \leq v.score$. We notice that after this operation, no instruction can change the value of $v.score$ (Lemma 5.19. □

Lemma 5.25. $(i^-)u.score = (i^+)u.score$ *unless u sends or receives a successful FLIP in round i.*

Proof. u changes its $score$ value only by executing `ADOPT_PARENT(`m_u`)` or `ADOPT_CHILD(`m_u`)`. Both instructions that changes $u.score$ value in these procedures (Algorithm 7, line 9, 16) are conditioned by $m_u.action = FLIP$. □

Lemma 5.26. *A node u changes $u.score$ at most once during a round.*

Proof. A node sends at most one FLIP message during a round. A node receives at most one FLIP message during a round (Lemma 5.19). Either a node receives a FLIP, sends one, or it does not receive and does not send a FLIP during a given round (Lemma 5.20). So, according to Lemma 5.25, a node changes $u.score$ at most once during a round. □

Lemma 5.27. *Before each round, the set of scores is a permutation of the set of identifiers.*

Proof. After the initialization in each node u, $u.score = u.ID$. A node u changes its score only by executing `ADOPT_PARENT()` or `ADOPT_CHILD()`. We will do a proof by induction. We assume at the beginning of round i, the set of scores is a a permutation of the set of indentifiers. We have for any node u, $mu.score = (i^-)u.score$.

According to Lemma 5.25, only a node sending or receiving a successful FLIP may change its $score$ value. Assume that the node u changes its $score$ value during round i. Without lost of generality, we assume u sends the successful FLIP to a node v in round i.

By hypothesis, u changes its $score$ to $(i^-)v.score$ during the execution of `ADOPT_PARENT()` in round i. We have $(i^-)u.score\ geq(i^-)v.score$. v executes the procedure `ADOPT_CHILD(`mu`)` at line 16 in round i. This procedure sets the current score of v to $max(v.score, mu.score)$, as $mu.score = (i^-)u.score$. After the execution of this instruction, we have $v.score = (i^-)u.score$.

According lemma 5.26, we have $(i^+)v.score = (i^-)u.score$ and $(i^+)u.score = (i^-)v.score$. □

Correctness of the resulting forest after mergings

In lemmas 5.31 and 5.32, we establish that if u sends a successful SELECT to v in round i either $(i^-)v.status = T$ or $(i^-)v.parent.status = T$. In the first case, we have $(i^-)u.score < (i^-)v.score$, and in the second case, we have $(i^-)u.score < (i^-)v.parent.score$. Let ch be a series of nodes u_0, u_1, u_2 such that $(i^+)u_j.parent = u_{j+1}$ and such that u_0 sends a successful SELECT to u_1 during the round i. As a ch's subchain of nodes having strictly increasing scores at the beginning of round i may be built: ch has not loop. So ch ends by a node having a token: all nodes on that chain are valid.

Lemma 5.28. *If v sends a message containing T in round i and $(i^+)v.status = N$, let $w = (i^+)v.parent$, then $(i^+)w.status = T$.*

Proof. If v sends a message containing T in round i, then $(i^+)v.status = T$. If $(i^+)v.status = N$, then v has executed `ADOPT_PARENT()` in round i, because it is the only procedure that sets $v.status$ to N. v executes `ADOPT_PARENT()` only if it has sent a FLIP message m_v to a node w ($m_v.action \neq$ SELECT because $m_v.senderStatus = T$), and if w has received the message m_v (*reciprocal reception property*). At the reception of m_v by w, w executes `BECOME_ROOT()` (line 16) which sets $w.status$ to T and from this line until the end of the round no instruction can change $w.status$ to N. So $(i^+)w.status = T$.

At the execution of `ADOPT_PARENT()` by v, v sets $v.parent$ to w. After this instruction there is only `BECOMES_ROOT()` that can modifie the value of $v.parent$, and which is conditioned by the reception of a FLIP message. According to lemma 5.20 v cannot call `BECOMES_ROOT()` because it cannot receive a FLIP message. So $w = (i^+)v.parent$.

So, if v sends a message containing T in round i and $(i^+)v.status = N$, and $w = (i^+)v.parent$, then $(i^+)w.status = T$. □

Lemma 5.29. *If v sends a message containing T in round i and $(i^+)v.status = N$, let $w = (i^+)v.parent$, then $(i^+)w.score \geq (i^-)v.score$.*

Proof. We have $(i^-)v.status = T$ because in round $i-1$, $v.status$ cannot be modified after the execution of `PREPARE_MESSAGE()`. If $(i^-)v.children \neq \emptyset$ then v sends a FLIP message to one of its children, named u, in round i. Either $(u, v) \in E_i$, then $(i^+)u.parent = v$, $(i^+)v.status = T$ and $(i^+)u.score \leq (i^-)v.score$ (see Lemmas 5.22 and 5.24). Otherwise $(i^+)v.status = T$. □

Lemma 5.30. *If u sends a successful SELECT to v in round i then $((i-1)^-)v.status = T$.*

Proof.
Node u prepared a SELECT message to v in round $i-1$, thus it had $u.contender = v$, which implies it received from v a message containing T. We have then $((i-1)^-)v.status = T$ because after the execution of `PREPERE_MESSAGE()` by v in round $i-2$, $v.status$ cannot be changed.

□

Lemma 5.31. *If u sends a successful SELECT to v in round i and $(i^-)v.status = T$, then $(i^-)u.score < (i^-)v.score$.*

Proof.
By Lemma 5.30 $((i-1)^-)v.status = T$. Then Lemmas 5.8 and 5.25 respectively imply that $(i^-)u.score < ((i-1)^-)v.score$ and $((i-1)^-)v.score = (i^-)v.score$.
□

Lemma 5.32. *If u sends a successful SELECT to v in round i and $(i^-)v.status = N$, then let $w = (i^-)v.parent$. It holds that $(i^-)w.status = T$ and $(i^-)u.score < (i^-)w.score$.*

Proof.
By Lemma 5.30 we have $((i-1)^-)v.status = T$. Then Lemmas 5.8 and 5.29 respectively imply that $(i^-)u.score < ((i-1)^-)v.score$ and $((i-1)^-)v.score \leq (i^-)w.score$. Lemma 5.28 implies that $(i^-)w.status = T$.
□

Lemma 5.33 (Cancellation)**.** *If u sends a failed FLIP or SELECT in round i, then $(i^+)u.status = T$.*

Proof. By lemma 5.6, we have $(i^-)u.status = T$. v did not receive the message from u implies that $(u, v) \notin E_i$. So, in round i, $v \notin u.neighbors$ (u did not receive the message from v). Only during the execution of `ADOPT_PARENT()`, called in line 9, u can change its $status$ to N. This procedure is not executed during the round i. □

Lemma 5.34 (Conservation)**.** *If $(i^-)u.status = T$ and u does not send a FLIP or SELECT in round i, then $(i^+)u.status = T$.*

Proof. By lemma 5.5, u does not execute the procedure `ADOPT_PARENT()` during the round i. u can set `status` variable to N only if it executes `ADOPT_PARENT()`. □

Lemma 5.35. *If $(i^-)u.status = T$ and u does not send a successful SELECT in round i, then u is valid after the round i.*

Proof. According to Lemma 5.23, after the successful sending of a FLIP message in round i, u is valid at the end of round i. If u sends a failed SELECT or a failed FLIP then u is valid after the round i by Lemma 5.33. otherwise, u did not send a SELECT or a FLIP during the round: it is also valid at the end of the round by Lemma 5.34. □

Lemma 5.36. *If a node sends a successful SELECT in round i, then it is valid at the end of round i.*

Proof. Let S be the set of nodes that send a successful SELECT in round i and are not valid at the end of round i. We will prove, by contradiction, that S is empty. Assume S is non-empty and consider the node in S that had the largest score at the beginning of round (say, node u). Such a node exists by Lemma 5.27. We will prove that u is valid after the round, which is a contradiction. Let v be the recipient of u's successful SELECT. By Lemma 5.21 $(i^+)u.parent = v$, thus is enough to show that v is valid after round i to get our contradiction. Let us examine both cases whether $(i^-)v.state = T$ or N.

If $(i^-)v.status = T$, then either v also sends a successful SELECT in round i, or it does not. If it does not, then it is valid after round i (Lemma 5.35). If it does, then it must be valid otherwise u is not maximal in S (Lemma 5.31).

If $(i^-)v.status = N$, then let $w = (i^-)v.parent$. Two cases are considered, whether $(v, w) \in E_i$ or not. If $(v, w) \notin E_i$ then $(i^+)v.status = T$ because the condition forces u to call the procedure `BECOME_ROOT()` in line 7 which makes it take the status T. After, u can takes the status N, only during the execution of the procedure `ADOPT_PARENT()` in line 9. This procedure is called by u only if u did send a FLIP or a SELECT at the beginning of round i by lemma 5.5. By Lemma 5.6, this cannot happen. Thus v is valid after round i. If $(v, w) \in E_i$, we use the fact that $(i^-)w.status = T$ (Lemma 5.28) to apply the same idea as we did above: either w also sends a successful SELECT in round i, or it does not. If it does not, then it is valid after round i (Lemma 5.35). If it does, then it must be valid otherwise u is not maximal in S (Lemma 5.32). □

Correctness of resulting forest

Lemma 5.37. *If $(i^-)u.status = T$ then u is valid after round i.*

Proof. According to Lemma 5.36, after the successful sending of a SELECT message in round i, u is valid at the end of round i. According to Lemma 5.23, after the successful sending of a FLIP message in round i, u is valid at the end of round i. If u sends a failed SELECT or a failed FLIP then u is valid after the round by Lemma 5.33. In otherwise, u is also valid the round by Lemma 5.34.

□

Theorem 5.2 (Resulting forest correctness)**.** *If all nodes are valid at the beginning of the round i, then all nodes are valid after round i.*

Proof. Assume that a node v is invalid after round i. According to Lemma 5.37, $(i^-)v.status = N$.

Let $u_0, u_1, u_2, ..., u_k$ be the finite series of nodes such that for $j \in [0, k-1]$, $(i^-)u_j.parent = u_{j+1}$, $(i^-)u_k.status = T$, and $u_0 = v$. This series exists because u is valid at the beginning of round i.

Let $u'_1, u'_2, ...,$ be the infinite series of nodes such that for all $j \geq 1$ $(i^+)u'_j.parent = u_{j+1}$, and $(i^+)v.parent = u'_1$. This series exists because v is invalid (by hypothesis).

According to Lemma 5.12, $j \in [1, k]$, $u_j = u'_j$. According to Lemma 5.37, u_k is valid. So all nodes of the series $u_0, u_1, u_2, ..., u_k$ are valid. There is a contradiction.

□

5.6 Simulation on Real World Traces (Infocomm 2006)

We verified the applicability of our algorithm to real world situations. The algorithm was implemented in the JBotSim simulator (Casteigts [2013]) and tested upon the `Infocomm06` dataset (Scott *et al.* [2006]). This dataset is a record of the possible

interactions between people during the Infocomm'06 conference. The resulting graph has the following characteristics: the number of nodes is 78 and the average node degree is 1.3. It should also be noted that an edge can appear at any time but the fact that it is still present is thereafter only tested every 120 seconds; this means that the presence time of an edge is a multiple of 120 seconds. Two cases were considered, based on the number of rounds one can assume to occur per second. The results show the average number of trees per connected component, averaged over 100 runs. In the first case (Figure 5.10), we assume that 10 rounds can be performed per second, which seems reasonable, yet optimistic. In the second case,

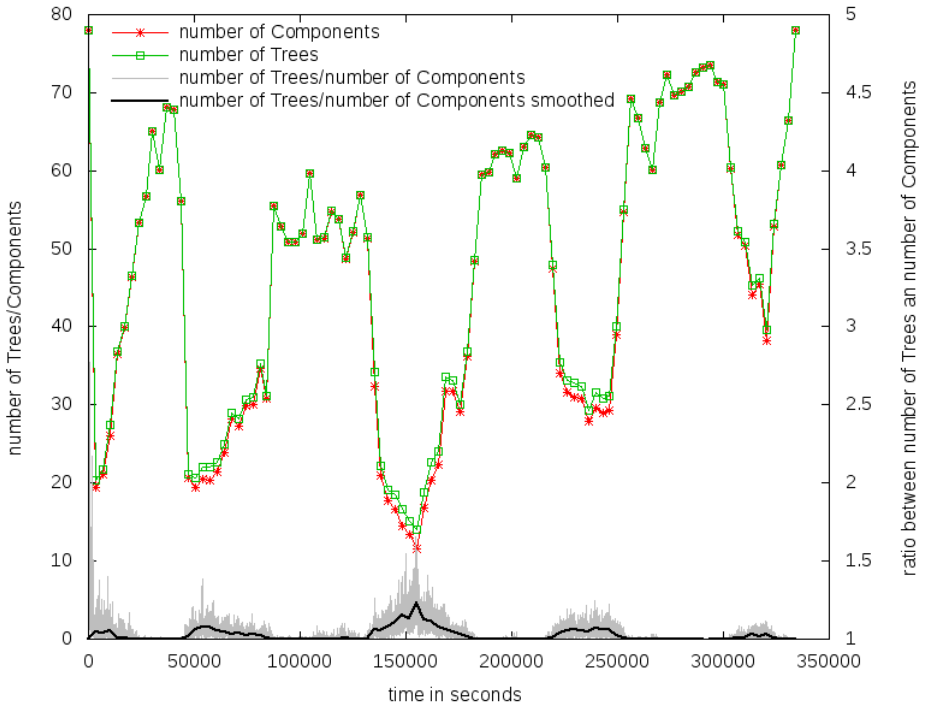


Figure 5.10: Number of roots per connected components, assuming 10 rounds per second.

we lower our expectations by assuming that only one round can be performed per second (Figure 5.11). These results show that the number of trees per connected component, averaged over time, is very close to 1 (about 1.027 in the first case, and 1.080 in the second case). Furthermore, the algorithm achieves an optimal configuration of a single spanning tree per connected component about 47% of the time in the first case (32.68% in the second case), which is encouraging. These results also validate the relevance of our algorithm in real-world scenarios.

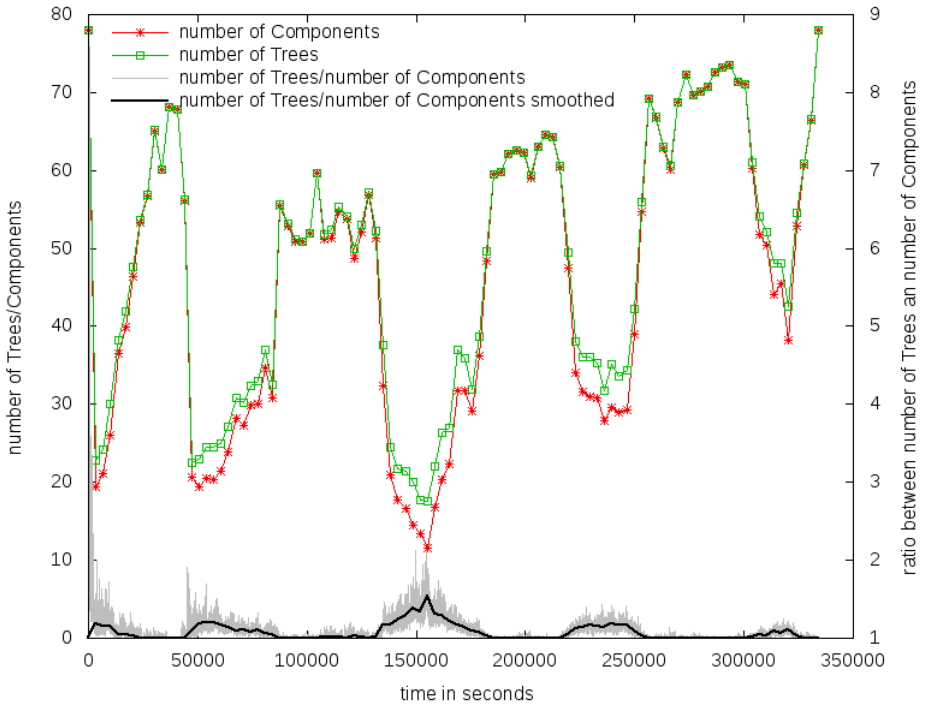


Figure 5.11: Number of roots per connected components, assuming only 1 round per second.

Conclusion

We presented in this chapter an algorithm that strives to maintain a forest of spanning trees in highly dynamic graphs, without any kind of assumption on the rate of changes. Our algorithm is the adaptation of a coarse-grain interaction algorithm (Casteigts *et al.* [2013a]) to the synchronous message passing model (for dynamic graphs). While the high-level principles of the coarse-grain variant are preserved, the new algorithm turns out to be significantly more complex. In particular, it involves a new technique that consists of maintaining a distributed permutation of the set of all nodes IDs throughout the execution. The algorithm also inherits the properties of its original variant: It relies on purely localized decisions, for which no global information is ever collected at the nodes, and yet it maintains a number of critical properties whatever the frequency and scale of the changes. In particular, the graph remains always covered by a spanning forest in which 1) no cycle can ever appear, 2) every node belongs to a tree, and 3) after an arbitrary number of edge disappearance, all maximal subtrees immediately restore exactly one token (at their root). These properties are ensured whatever the dynamics, even if it keeps going for an arbitrary long period of time. Optimality is not the focus here, however the number of tree per components – the metric of interest here – eventually converges to one if the graph stops changing (which is never expected to happen, though). The

algorithm correctness is proven and its behavior is tested through experimentation.

Conclusion

In this thesis we dealt with dynamic graphs classes and distributed algorithms in dynamic graphs. We contributed to the automation of dynamic graph classification. The proposed methods are part of a more general framework used to verify the correct behavior of a distributed algorithm and its appropriateness to a mobility model that represents an analytical alternative to the classical approach of simulations (executing the algorithm on instances of a mobility models or real network traces). Contrary to simulations whose results are difficult to reproduce and generalize, the automatic classification process can test whether a set of algorithms (whose topological requirements are already defined) can be executed successfully in a given dynamic graph by testing its membership to a class characterized by conditions necessary or sufficient for the success of theses algorithms.

We provided in this work new strategies and techniques allowing one to test the membership of a given dynamic graph to a given class and to compute parameters on two categories of temporal properties characterized by two types of problems, minimization and maximization problems. Another purpose of this work was to attempt to understand the case where the dynamics of the graph is unrestricted through the problem of maintaining a forest of spanning trees in a model where no assumptions about the dynamics of the graph is made. These contributions are summarized below.

Classification Algorithms for Specific Classes

As a first contribution we proposed strategies to test the membership of a given dynamic graph to two specific classes.

We addressed, in Chapter 2, the problem of testing whether a given dynamic graph is temporally connected, *i.e.* a *journey* exists between all pairs of nodes. We represented the the dynamic graph $\mathcal{G}$ as a sequence of graphs (untimed evolving graph). We studied two cases, the case of strict journeys where a single edge can be crossed in a step G_i and the case of non-strict journeys where an unlimited number of edges can be crossed in a same step G_i. In the case of *strict* journeys, we investigated algorithmic solutions based on the computation of the transitive closure of

journeys. We compared the complexity of our algorithm to that of the adaptation of the other solutions: the solution presented in Whitbeck *et al.* [2012] is always more costly (keep in mind that it solves a more general problem), the other solution presented in Bui-Xuan *et al.* [2003] is more or less costly depending on the interplay between instant density and cumulated density. Our solution is shown to be relevant for sparse mobility scenario (e.g. robots or UAVs exploring an area) where the number of neighbors at a given time is low, though many nodes can be seen over the whole execution. In the case of *non-strict* journeys, for which no algorithm is known, we showed that some pre-processing of the input graph allows us to re-use the same algorithm than before. By chance, these operations have the same cost as that of the application of the first algorithm, which implies that the second problem is not more difficult than the first.

In Chapter 3, a second problem we studied is that of testing whether a given dynamic graph $\mathcal{G}$ is T-interval connected. We also considered the related problem of finding the largest T for which a given $\mathcal{G}$ is T-interval connected. We assumed that the dynamic graph $\mathcal{G}$ is a sequence of graphs. We investigated algorithmic solutions that use two elementary operations, *binary intersection* and *connectivity testing*. For this problems, we developed efficient algorithms that use linear number of elementary operations in the length of the graph, which asymptotically matches the lower bound. We presented PRAM algorithms that show that both problems can be solved efficiently in parallel and we provided online versions of the algorithms that use a constant number of elementary operations per graph received. We also presented dynamic versions of the online algorithms that report connectivity based on recent network history.

Generic Framework for Testing Properties

The presented framework used to solve the problem of testing T-interval connectivity is generalized, in Chapter 4, to solve other problems. The generic framework and algorithm are proposed to solve a category of problems called minimization problems i.e. finding the smallest value corresponding to a parameter of a property (as the problem of finding T for which a graph is T-interval connected, which in this case is a maximization problem). We illustrate the use of our generic framework by applying the generic algorithm to solve the three problems of finding: the temporal diameter, the round trip temporal diameter of a given dynamic graph $\mathcal{G}$ and a bound on the realization of its footprint.

We focused on algorithms using only two elementary operations *composition* and *test* operations. This approach is suitable for a high-level study when the details of changes between successive graphs in a sequence are arbitrary. If the evolution of the dynamic graph is constrained in some ways (e.g., bounded number of changes

between graphs), then one could benefit from the use of more sophisticated data structures to lower the complexity of the problem.

Distributed Algorithms in Arbitrary Dynamic graphs

In Chapter 5 we attempted to understand what can still be computed (and guaranteed) when no assumptions are made on the graph dynamics: neither on the rate of change, nor on their simultaneity, nor on global connectivity. In other words, we do not assume that the graph belongs to any of presented classes. In this seemingly chaotic context, we presented an algorithm that strives to maintain a forest few trees per components as possible, while always guaranteeing some properties. The algorithm relies on purely localized decisions, for which no global information is ever collected at the nodes, and yet it maintains a number of critical properties whatever the frequency and scale of the changes. In particular, the graph remains always covered by a spanning forest in which 1) no cycle can ever appear, 2) every node belongs to a tree, and 3) after an arbitrary number of edge disappearance, all maximal subtrees immediately restore exactly one token (at their root). These properties are ensured whatever the dynamics, even if it keeps going for an arbitrary long period of time.

Open Questions and Future Works

A natural extension of our investigation would be a similar study for other classes and properties of dynamic graphs, as identified in Casteigts *et al.* [2012]. Other strategies and algorithms could be proposed to test other specific classes and the use of the proposed generic framework for the resolution of other minimization problems could also be interesting to aim for.

Distributed algorithms for other classification problems, in which a node in the graph only sees its local neighbourhood, would be of interest. In this case, each node must compute an output that 1) answers a decision question like deciding whether the dynamic graph or a part of it has a given temporal property or not, e.g. is there a bound on the realization of the footprint graph (bound on the reappearance of edges), or 2) represents a value of a metric or a parameter e.g. what is the bound on the footprint graph realization? This kind of algorithms allows a distributed algorithm executed in a higher layer, to collect information on the dynamic of the graph and adapt its behavior based on this information. For example, the algorithm for the maintenance of a spanning forest in highly dynamic networks, that we presented, can depend on an algorithm in a lower layer that informs each node about the bound of communication links reappearance in the local network (adjacent links). This allows, for example, to avoid breaking a spanning tree and regenerating a token if the links disappear for a period considered as relatively short. Another example is that of T-Interval connectivity. Distributed versions of the dynamic algorithms proposed

for T-interval connectivity could be used to supplement the information available to distributed Internet routing protocols such as OSPF (Open-Shortest Path First) which are used to construct routing tables. Our dynamic algorithms have constant amortized complexity, and distributed versions with constant amortized complexity could provide real-time information about network connectivity to OSPF.

Beside this extensions of this thesis work, the characterization of new necessary or sufficient conditions for distributed problems (and the discovery of new corresponding classes of dynamic graphs) seems a promising avenue for future work.

Bibliography

ABBAS, Sheila, MOSBAH, Mohamed and ZEMMARI, Akka, 2006. Distributed computation of a spanning tree in a dynamic graph by mobile agents. Dans *Proc. of IEEE Int. Conference on Engineering of Intelligent Systems (ICEIS)*, pages 1–6. doi:10.1109/ICEIS.2006.1703205.

ALDOUS, David and FILL, Jim, 2002. Reversible markov chains and random walks on graphs.

AWERBUCH, Baruch, CIDON, Israel and KUTTEN, Shay, 2008. Optimal maintenance of a spanning tree. *J. ACM*, 55(4):18:1–18:45. doi:10.1145/1391289.1391292.
URL http://doi.acm.org/10.1145/1391289.1391292

AWERBUCH, Baruch and EVEN, Shimon, 1984a. Efficient and reliable broadcast is achievable in an eventually connected network. Dans *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 278–281. ACM.

AWERBUCH, Baruch and EVEN, Shimon, 1984b. Efficient and reliable broadcast is achievable in an eventually connected network(extended abstract). Dans *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, pages 278–281. ACM, New York, NY, USA. ISBN 0-89791-143-1. doi:10.1145/800222.806754.
URL http://doi.acm.org/10.1145/800222.806754

BAALA, Hichem, FLAUZAC, Olivier, GABER, Jaafar, BUI, Marc and EL-GHAZAWI, Tarek, 2003. A self-stabilizing distributed algorithm for spanning tree construction in wireless ad hoc networks. *Journal of Parallel and Distributed Computing*, 63:97–104.

BAR-ILAN, Judit and ZERNIK, Dror, 1989. Random leaders and random spanning trees. Dans Jean-Claude Bermond and Michel Raynal, rédacteurs, *Workshop on Distributed Algorithms (WDAG)*, tome 392 de *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin Heidelberg. ISBN 978-3-540-51687-3. doi:10.1007/3-540-51687-5_27.
URL http://dx.doi.org/10.1007/3-540-51687-5_27

BARJON, M., CASTEIGTS, A., CHAUMETTE, S., JOHNEN, C. and NEGGAZ, Y., 2014a. Maintaining a spanning forest in highly dynamic networks: The synchronous case. Dans *18th Int. Conference on Principles of Distributed Systems (OPODIS)*, numéro 8878 dans Lecture Notes in Computer Science, pages 277–292.

BARJON, Matthieu, CASTEIGTS, Arnaud, CHAUMETTE, Serge, JOHNEN, Colette and NEGGAZ, Yessin M., 2014b. Testing temporal connectivity in sparse dynamic graphs. *CoRR*, abs/1404.7634:8p. A French version appeared in Proc. of ALGOTEL (2014).

BARJON, Matthieu, CASTEIGTS, Arnaud, CHAUMETTE, Serge, JOHNEN, Colette and NEGGAZ, Yessin M., 2014c. Un algorithme de test pour la connexité temporelle dans les graphes dynamiques de faible densité. Dans *ALGOTEL 2014 - 16èmes Rencontres Francophones sur les Aspects Algorithmiques de Télécommunications ALGOTEL*.

BERNARD, Thibault, BUI, Alain and SOHIER, Devan, 2013. Universal adaptive self-stabilizing traversal scheme: Random walk and reloading wave. *J. Parallel Distrib. Comput.*, 73(2):137–149.

BHADRA, Sandeep and FERREIRA, Afonso, 2003. Complexity of connected components in evolving graphs and the computation of multicast trees in dynamic networks. Dans *Second international conference on Ad-Hoc, Mobile, and Wireless Networks (ADHOCNOW)*, LNCS 2865, pages 259–270. Springer.

BUI-XUAN, Binh-Minh, FERREIRA, Afonso and JARRY, Aubin, 2003. Computing shortest, fastest, and foremost journeys in dynamic networks. *Int. J. of Foundations of Computer Science*, 14(02):267–285. doi:10.1142/S0129054103001728. URL http://www.worldscientific.com/doi/abs/10.1142/S0129054103001728

BURGARD, Wolfram, MOORS, Mark, FOX, Dieter, SIMMONS, Reid and THRUN, Sebastian, 2000. Collaborative multi-robot exploration. Dans *IEEE International Conference on Robotics and Automation (ICRA)*, pages 476–481. IEEE.

BURMAN, Janna and KUTTEN, Shay, 2007. Time optimal asynchronous self-stabilizing spanning tree. Dans Andrzej Pelc, rédacteur, *Distributed Computing*, tome 4731 de *Lecture Notes in Computer Science*, pages 92–107. Springer Berlin Heidelberg. ISBN 978-3-540-75141-0. doi:10.1007/978-3-540-75142-7_10. URL http://dx.doi.org/10.1007/978-3-540-75142-7_10

CASTEIGTS, A., 2006. Model driven capabilities of the DA-GRS model. Dans *Proc. of 1st Intl. Conference on Autonomic and Autonomous Systems (ICAS'06)*, pages 24–32. IEEE Computer Society, Washington, DC, USA. ISBN 0-7695-2653-5. doi:http://dx.doi.org/10.1109/ICAS.2006.35.

CASTEIGTS, A., CHAUMETTE, S. and FERREIRA, A., 2009. Characterizing topological assumptions of distributed algorithms in dynamic networks. Dans *Proc. of SIROCCO*, pages 126–140. Springer, Piran, Slovenia. (Full version in *CoRR, abs/1102.5529*).

CASTEIGTS, A., FLOCCHINI, P., QUATTROCIOCCHI, W. and SANTORO, N., 2012. Time-varying graphs and dynamic networks. *International Journal of Parallel, Emergent and Distributed Systems*, 27(5):387–408. doi:10.1080/17445760.2012.668546.

CASTEIGTS, Arnaud, 2013. The JBotSim library. *CoRR*, abs/1001.1435. See also the project website at http://jbotsim.sourceforge.net.

CASTEIGTS, Arnaud, CHAUMETTE, Serge, GUINAND, Frédéric and PIGNÉ, Yoann, 2013a. Distributed maintenance of anytime available spanning trees in dynamic networks. Dans *Proceedings of 12th conf. on Adhoc, Mobile, and Wireless Networks (ADHOC-NOW)*, tome 7960 de *Lecture Notes in Computer Science*, pages 99–110. Wroclaw, Poland.

CASTEIGTS, Arnaud, FLOCCHINI, Paola, GODARD, Emmanuel, SANTORO, Nicola and YAMASHITA, Masafumi, 2013b. Expressivity of time-varying graphs. Dans *19th International Symposium on Fundamentals of Computation Theory (FCT)*. Liverpool, United Kingdom.

CASTEIGTS, Arnaud, FLOCCHINI, Paola, MANS, Bernard and SANTORO, Nicola, 2014. Measuring temporal lags in delay-tolerant networks. *IEEE Transactions on Computers*, 63(2):397–410.

CASTEIGTS, Arnaud, KLASING, Ralf, NEGGAZ, Yessin M. and PETERS, Joseph G., 2015a. Efficiently testing t-interval connectivity in dynamic graphs. Dans *CIAC 2015-9th Int. Conference on Algorithms and Complexity CIAC*, Lecture Notes in Computer Science. Paris, France.

CASTEIGTS, Arnaud, KLASING, Ralf, NEGGAZ, Yessin M. and PETERS, Joseph G., 2015b. Tester efficacement la T-intervalle connexité dans les graphes dynamiques. Dans *ALGOTEL 2015 - 17èmes Rencontres Francophones sur les Aspects Algorithmiques de Télécommunications ALGOTEL*, ALGOTEL 2015 - 17èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications. Beaune, France.
URL https://hal.archives-ouvertes.fr/hal-01147110

FERREIRA, Afonso, 2002. On models and algorithms for dynamic communication networks: The case for evolving graphs. Dans *Proc. of 4e rencontres francophones sur les Aspects Algorithmiques des Télécommunications (ALGOTEL)*.

FLOCCHINI, P., MANS, B. and SANTORO, N., 2013. On the exploration of time-varying networks. *Theoretical Computer Science*, 469:53–68. doi:10.1016/j.tcs.2012.10.029.
URL http://dx.doi.org/10.1016/j.tcs.2012.10.029

GIBBONS, Alan and RYTTER, Wojciech, 1988. *Efficient parallel algorithms*. Cambridge University Press. ISBN 978-0-521-38841-2.

ISRAELI, Amos and JALFON, Marc, 1990. Token management schemes and random walks yield self-stabilizing mutual exclusion. Dans *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 119–131. ACM.

JAIN, S., FALL, K. and PATRA, R., 2004. Routing in a delay tolerant network. Dans *Proc. of SIGCOMM*, pages 145–158.

JÁJÁ, Joseph, 1992. *An Introduction to Parallel Algorithms*. Addison-Wesley. ISBN 0-201-54856-9.

JEA, David, SOMASUNDARA, Arun and SRIVASTAVA, Mani, 2005. Multiple controlled mobile elements (data mules) for data collection in sensor networks. Dans *First IEEE international conference on Distributed Computing in Sensor Systems(DCOSS'2005)*, pages 244–257. Springer.

KEMPE, David, KLEINBERG, Jon and KUMAR, Amit, 2000. Connectivity and inference problems for temporal networks. Dans *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*, STOC '00, pages 504–513. ACM, New York, NY, USA. ISBN 1-58113-184-4. doi:10.1145/335305.335364.
URL http://doi.acm.org/10.1145/335305.335364

KOSSINETS, Gueorgi, KLEINBERG, Jon and WATTS, Duncan, 2008. The structure of information pathways in a social communication network. Dans *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 435–443. ACM.

KRAVCHIK, Alex and KUTTEN, Shay, 2013. Time optimal synchronous self stabilizing spanning tree. Dans Yehuda Afek, rédacteur, *Distributed Computing*, tome 8205 de *Lecture Notes in Computer Science*, pages 91–105. Springer Berlin Heidelberg. ISBN 978-3-642-41526-5. doi:10.1007/978-3-642-41527-2_7.
URL http://dx.doi.org/10.1007/978-3-642-41527-2_7

KUHN, F., LYNCH, N. and OSHMAN, R., 2010. Distributed computation in dynamic networks. Dans *Proc. of STOC*, pages 513–522. ACM, Cambridge, USA.

LITOVSKY, Igor, MÉTIVIER, Yves and SOPENA, Éric, 1999. Handbook of graph grammars and computing by graph transformation. chapitre Graph Relabelling

Systems and Distributed Algorithms, pages 1–56. World Scientific Publishing Co., Inc., River Edge, NJ, USA. ISBN 9-810240-21-X.
URL http://dl.acm.org/citation.cfm?id=320647.320649

LITOVSKY, Igor, METIVIER, Yves and SOPENA, Eric, 2001. Graph relabelling systems and distributed algorithms. Dans *Handbook of graph grammars and computing by graph transformation*. Citeseer.

MARCHAND DE KERCHOVE, Florent and GUINAND, Frédéric, 2012. Strengthening Topological Conditions for Relabeling Algorithms in Evolving Graphs. Working paper or preprint.
URL https://hal.archives-ouvertes.fr/hal-00743565

O'DELL, R. and WATTENHOFER, R., 2005. Information dissemination in highly dynamic graphs. Dans *Proc. of DIALM-POMC*, pages 104–110. ACM, Cologne, Germany. ISBN 1-59593-092-2.

SCOTT, James, GASS, Richard, CROWCROFT, Jon, HUI, Pan, DIOT, Christophe and CHAINTREAU, Augustin, 2006. Crawdad trace cambridge/haggle/imote/infocom (v. 2006-01-31). *http://crawdad.cs.dartmouth.edu/cambridge/haggle/imote/infocom*.

SHAH, Rahul C, ROY, Sumit, JAIN, Sushant and BRUNETTE, Waylon, 2003. Data mules: Modeling and analysis of a three-tier architecture for sparse sensor networks. *Ad Hoc Networks*, 1(2):215–233.

VIARD, Tiphaine, LATAPY, Matthieu and MAGNIEN, Clémence, 2016. Computing maximal cliques in link streams. *Theoretical Computer Science*, 609 Part 1:245–252. doi:10.1016/j.tcs.2015.09.030.
URL https://hal.archives-ouvertes.fr/hal-01112627

WHITBECK, John, DIAS DE AMORIM, Marcelo, CONAN, Vania and GUILLAUME, Jean-Loup, 2012. Temporal reachability graphs. Dans *Proc. of MOBICOM*, pages 377–388. ACM.