Thesis submitted to obtain the degree of doctor of philosophy from

Sorbonne Université

École doctorale EDITE de Paris (ED130)
Informatique, Télécommunication et Électronique

Laboratoire d'Informatique de Paris 6 (LIP6)
Thalès SIX - ThereSIS

Temporal Connectivity and Path Computation for Stream Graph

–

Connectivité Temporelle et Calcul de Chemins dans les Stream Graphs

---

Léo RANNOU

Presented on November 9th, 2020 in front of a jury composed of:

Reviewers:

- **Christophe CRESPELLE**, Associate professor at Université Claude Bernard Lyon 1, LIP
- **Sylvain PEYRONNET**, Professor at IX-LABS,

Examiners:

- **Lionel LACASSAGNE**, Professor at Sorbonne Université, LIP6
- **François SAUSSET**, Research Engineer at Thales, ThereSIS

Under the supervision of:

- **Clémence MAGNIEN**, Research Director at CNRS, LIP6
- **Matthieu LATAPY**, Research Director at CNRS, LIP6

Remerciements

Je tiens à remercier Matthieu Latapy et Clémence Magnien, pour leur accompagnement et conseils tout au long de cette thèse. Les conditions de travail au sein de l'équipe Complex Networks étaient proches de l'idéal, je tiens donc à en remercier tous les membres présents et passés qui y ont participé.

L'équipe data science de ThereSIS qui n'a fait que s'agrandir, chaque arrivée contribuant positivement à l'ambiance générale. Pour leur bonne humeur et leur sympathie, j'en remercie tous les membres passés et actuels.

A tout ceux qui m'ont encouragé à rédiger quand j'en avais moyennement envie, un grand merci, cela a parfois été salvateur.

Et, bien sur, je remercie tout particulièrement Éléonore qui s'est farcie la fastidieuse relecture de ce manuscrit et m'a moralement accompagné ces dernières années.

Abstract

Keywords: stream graphs, temporal networks, time-varying graphs, dynamic graphs, dynamic networks, interactions, graphs, networks, connected components, temporal paths, algorithms, link streams

For a long time, structured data and temporal data have been analysed separately. Many real world complex networks have a temporal dimension, such as contacts between individuals or financial transactions. Graph theory provides a wide set of tools to model and analyze static connections between entities. Unfortunately, this approach does not take into account the temporal nature of interactions. Stream graph theory is a formalism to model highly dynamic networks in which nodes and/or links arrive and/or leave over time. The number of applications of stream graph theory has risen rapidly, along with the number of theoretical concepts and algorithms to compute them. Several theoretical concepts such as connected components and temporal paths in stream graphs were defined recently, but no algorithm was provided to compute them. Moreover, the algorithmic complexities of these problems are unknown, as well as the insight they may shed on real-world stream graphs of interest.

In this thesis, we present several solutions to compute notions of connectivity and path concepts in stream graphs. We also present alternative representations - data structures designed to facilitate specific computations - of stream graphs. We provide implementations and experimentally compare our methods in a wide range of practical cases. We show that these concepts indeed give much insight on features of large-scale datasets. Straph, a python library, was developed in order to have a reliable library for manipulating, analysing and visualising stream graphs, to design algorithms and models, and to rapidly evaluate them.

French Abstract

Les données structurelles et les données temporelles ont, pendant longtemps, été analysées séparément. De nombreux réseaux complexes contiennent une dimension temporelle, comme les contacts entre individus ou les transactions financières. La théorie des graphes fournit un large ensemble d'outils pour modéliser et analyser les connexions entre entités. Malheureusement, cette approche ne prend pas compte la nature temporelle des interactions. La théorie des *stream graphs* est un formalisme permettant de modéliser les réseaux dynamiques dans lesquels les nœuds et/ou les liens arrivent et/ou partent au fil du temps. Plusieurs concepts théoriques tels que les composantes connexes dans les *stream graphs* ont été définis récemment, mais aucun algorithme n'a été proposé pour les calculer. De plus, la complexité algorithmique de ces problèmes est inconnue, ainsi que les connaissances qu'ils peuvent apporter sur les *stream graphs* de terrain.

Dans cette thèse, nous proposons plusieurs solutions pour le calcul de notions de connectivité et de chemins dans les *stream graphs*. Nous présentons également des représentations alternatives - des structures de données conçues pour faciliter certains calculs - des *stream graphs*. Nous fournissons également des implémentations et comparons expérimentalement nos méthodes sur une grande variété de cas pratiques. Nous montrons que ces concepts apportent beaucoup d'informations sur les caractéristiques de ces ensembles de données. Straph, une bibliothèque python, a été développée afin de disposer d'une ressource fiable afin de manipuler, analyser et visualiser les *stream graphs*.

Résumé

Nos sociétés hyperconnectées fourmillent de réseaux, que ce soit dans les échanges (transactions financières, flux logistique) ou les télécommunications, s'immisçant dans tous les aspects de nos relations sociales (messagerie, mails, réseaux sociaux, applications de rencontres). Il est primordial de disposer d'outils permettant de modéliser la richesse de ces structures connectées en perpétuelle évolution. Quelle qu'en soit la perspective, comprendre et analyser ces objets complexes est devenu un enjeux majeur.

La science des réseaux est un domaine de recherche bien établi, nombre de problèmes susmentionnés ont déjà été abordés efficacement, en fournissant des algorithmes et des outils pour l'analyse et la modélisation des réseaux [14, 19, 104, 106, 72, 9]. Cependant, la science des réseaux a longtemps été limitée à des phénomènes statiques. Plus précisément, elle a été appliquée aux réseaux sans tenir compte de la nature temporelle intrinsèque de nombreux phénomènes. Si nous considérons l'exemple des relations et communications entre individus, il est impossible d'ignorer la nature temporelle des interactions humaines sans perdre de l'information. L'approche guidant notre travail consiste à introduire cette dimension temporelle dans la modélisation sous forme de réseaux. Le formalisme que nous utilisons dans cette thèse est celui des *stream graphs* [59]. De même que la théorie des séries temporelles fournit des outils mathématiques et une modélisation spécifique pour analyser les signaux sous forme de séquences continues de points de données, la théorie des *stream graphs* fournit des outils pour l'analyse et la modélisation de séquences continues de liens, ou de graphes. En outre, ce formalisme est conforme à la théorie des graphes: si un *stream graph* n'évolue pas, ne possède pas de dynamique, il est équivalent à un graphe et ses propriétés sont les mêmes que celles de ce graphe [59].

Le but de cette thèse est double : il s'agit de tirer profit du formalisme des *stream graphs* afin d'analyser la dynamique et la structure des données du monde réel ; et en retour de tirer profit de ces applications pratiques pour étendre et améliorer le formalisme, en particulier pour résoudre les défis algorithmiques associés à sa mise en œuvre sur des données massives.

Objectifs

- Résoudre les questions algorithmiques liées au calcul des concepts de connexité et de chemins dans les *stream graphs*, en particulier sur les ensembles de données massifs.
- Développer une bibliothèque python pour analyser et manipuler les *stream graphs*. Il s'agit de développer des structures de données spécifiques ainsi que d'utiliser des implémentations parallèles et/ou en *streaming* afin de faire passer à l'échelle les différentes méthodes sur des ensembles de données de terrain.
- Utiliser les concepts définis dans le formalisme pour analyser la structure et la dynamique des données du monde réel.

Dans la suite, nous allons présenter succinctement le contenu des différents chapitres de cette thèse ainsi que les principales contributions réalisées. Le lecteur peut se référer aux chapitres en anglais pour plus de détails.

Chapitre 1 : Modélisation de réseaux temporels par Stream Graphs

Dans cette partie nous présentons tout d'abord des concepts et propriétés élémentaires de la théorie des graphes ainsi que les concepts et propriétés équivalents pour les *stream graphs*. Nous introduisons les notations, les définitions et le matériel de base qui seront utilisés dans le cadre de cette thèse. Nous passons également en revue les différentes approches de l'état de l'art pour modéliser les réseaux temporels et soulignons leurs similitudes et leurs différences avec les nôtres. Enfin, nous montrons comment les *stream graphs* sont particulièrement bien adaptés pour modéliser des données structurelles dynamiques. Nous décrivons et présentons également les principales propriétés de certains ensembles, représentatifs et de différentes échelles, de données de terrain.

Chapitre 2 : Straph: Une bibliothèque Python pour les Stream Graphs

Contributions

- Une bibliothèque python open source pour la manipulation, la modélisation, l'analyse et la visualisation des *stream graphs* : Straph
- Des structures de données efficaces pour gérer les *stream graphs*
- Deux générateurs de *stream graphs* aléatoires à partir des modèles d'Erdős-Rényi et de Barabási-Albert

Straph est un paquet Python 3 open source, sous licence Apache 2.0, pour l'exploration

et l'analyse de *stream graphs* réels et artificiels. Cette bibliothèque fournit des structures de données spécifiques pour représenter différents types de *stream graphs*, des algorithmes pour calculer des propriétés et des mesures de base, des connecteurs pour divers formats de données ainsi que des générateurs similaires aux modèles Erdős-Rényi [30] et Barabási-Albert [7]. À long terme, nous espérons fournir l'équivalent de Networkx ou Networkit, en termes de fonctionnalités, pour les *stream graphs*. Straph peut être utilisé pour enseigner la théorie des *stream graphs*, illustrer des concepts particuliers ou encore mener des expériences. Cet outil peut aussi être utilisé par des utilisateurs ou des développeurs qui ne sont pas nécessairement des experts en programmation ou en théorie des *stream graphs*.

Dans ce chapitre, nous détaillons les paradigmes guidant le développement de Straph et nous donnons un aperçu de l'architecture de la bibliothèque ainsi que des structures de données employées. Nous présentons ensuite les nombreuses caractéristiques et possibilités de Straph et nous illustrons comment les utiliser en pratique. Toutefois, au moment de l'écriture de ces lignes, de nombreuses fonctionnalités doivent être rigoureusement testées et documentées. Straph n'est pas suffisamment mûr pour être utilisé en production.

Enfin, nous discutons des caractéristiques potentielles et futures. À l'avenir, nous avons l'intention d'étendre Straph afin de traiter différents types de *stream graphs* : possédant des liens pondérés, où emprunter un lien correspond à un temps de parcours, avec une structure bipartite. L'ajout d'autres algorithmes tels que la détection de communauté, les mesures de centralité, la prédiction de liens et de nœuds sont envisagés, afin de relever de nombreux défis pratiques.

Chapitre 3 : Connexité

Contributions

- Des algorithmes pour calculer les notions de connexité dans les *stream graphs*
- Une analyse de la connexité de *stream graphs* réels de grande taille

Le concept de connexité est fondamental en théorie des graphes. Il est d'usage de décomposer un graphe en ses composantes connexes distinctes. De nombreuses propriétés, qui impliquent le calcul de chemins ou de communautés, peuvent être calculées indépendamment sur chaque composante connexe, permettant ainsi l'exécution en parallèle de nombreuses méthodes.

Dans ce chapitre, nous présentons des notations et des définitions clés et introduisons des algorithmes, linéaires et polynomiaux, pour calculer les composantes faiblement et fortement connexes dans les *stream graphs*. Nous appliquons ensuite ces algorithmes à plusieurs ensembles de données du monde réel à grande échelle, afin d'étudier leurs performances en pratique et démontrer leur capacité à décrire de tels ensembles de

données.

Ces algorithmes peuvent traiter des flux de dizaines de millions d'événements et peuvent retourner les composantes connexes en *streaming*. Ils rendent les composantes connexes utilisables en pratique. À notre connaissance, c'est la première fois qu'une partition des nœuds temporels en composantes connexes est calculée à une telle échelle.

Chapitre 4 : Représentations Alternatives des Stream Graphs

Contributions

- Une représentation des données basée sur des concepts de connexité réduisant considérablement la complexité des calculs de requêtes d'accessibilité : la condensation d'un *stream graph*
- Une représentation alternative des données permettant un calcul parallèle efficace de nombreuses propriétés des *stream graphs* : le graphe orienté acyclique stable d'un *stream graph*
- Une méthode d'approximation accélérant le calcul de nombreuses méthodes en pratique tout en préservant les propriétés de connexité d'un *stream graph* : la Δ-approximation

Dans ce chapitre, nous présentons des représentations alternatives - des structures de données conçues pour faciliter certains calculs - des *stream graphs*. La principale motivation est de pouvoir effectuer certaines tâches de manière efficace (répondre à différents types de requêtes, faciliter le calcul de certaines mesures et/ou de propriétés).

Notre première représentation alternative, la condensation, utilise les composantes fortement connexes comme blocs de construction et réduit la complexité des requêtes d'accessibilité. Après avoir défini cette représentation, nous proposons une méthode pour calculer la condensation à partir d'un *stream graph*, basée sur un algorithme de connexité, défini précédemment. Ensuite, nous utilisons ses principales propriétés pour concevoir des algorithmes répondant aux requêtes d'accessibilité, en temps linéaire, et nous procédons à leur évaluation sur des *stream graphs* réels. Nous définissons la notion de graphe orienté acyclique stable et utilisons ses propriétés pour concevoir une méthode calcul parallèle efficace pour le calcul des nombreuses propriétés des *stream graphs*. Troisièmement, nous proposons une méthode d'approximation, la Δ-approximation, qui préserve la connexité d'un *stream graph* tout en diminuant le temps de calcul de plusieurs méthodes. Enfin, nous concluons en proposant des idées pour améliorer ces représentations de *stream graphs* et en suggérant d'autres applications potentielles.

Chapitre 5 : Chemins Temporels

Contributions

- Un algorithme générique en temps polynomial calculant tout type de chemin temporel optimal dans les *stream graphs* : le *L-algorithme*
- Des algorithmes en temps linéaire reposant sur la condensation d'un stream graph pour le calcul des chemins arrivant le plus tôt ainsi que des chemins les plus rapides.

Ce chapitre est dédié aux chemins, un concept majeur de la théorie des graphes, aux problèmes associés et aux algorithmes les résolvant. Les applications sont nombreuses et d'une importance capitale, comme le calcul du chemin le plus court, ou le plus rapide, d'une entité à une autre dans un réseau de communication. Des notions de chemin dans les réseaux temporels ou dynamiques ont été proposées [12, 47, 112, 109] ainsi que des algorithmes les calculant [109, 112, 78, 21, 96]. Cependant, la modélisation par *stream graphs* diffère des formalismes de réseaux temporels existants. Par conséquent, différentes notions de chemins de *stream graphs* ont été proposées dans [59] mais il reste à relever les défis algorithmiques qui en découlent.

Tout d'abord, après avoir détaillé les notions de chemins temporels dans les *stream graphs*, nous présentons différents types de problèmes de chemins temporels optimaux survenant dans les *stream graphs*. Nous montrons qu'un problème de chemin optimal peut être abordé comme un problème d'optimisation reposant sur une fonction objective et une autre de domination. Nous proposons ensuite une procédure unique, un algorithme polynomial générique - qui peut être considéré comme une généralisation du célèbre algorithme de Dijkstra - le *L-Algorithm*, pour calculer efficacement tout type de chemin temporel optimal.

Nous présentons également des algorithmes utilisant la condensation d'un *stream graph*, défini au chapitre précédent, pour réduire la complexité algorithmique. Nous montrons que ces algorithmes sont plus rapides ou équivalents à nos *L-Algorithmes* pour les problèmes de chemins arrivant le plus tôt et des plus rapides, au prix d'un prétraitement polynomial. Enfin, nous évaluons nos algorithmes sur quatorze ensembles de données du monde réel, démontrant leur efficacité pratique sur des *stream graphs* de dizaines de millions d'évènements.

Conclusion

Dans cette thèse, nous avons exploré des concepts généralisés, cohérents avec la théorie des graphes, afin de prendre en compte une dimension temporelle. Nous avons, en sus, établi et fourni des structures de données ainsi que leur implémentations afin d'analyser et de manipuler des *stream graphs* réels ou synthétiques.

Toutefois, le travail nécessaire pour parvenir à des algorithmes efficaces pour chaque

concept des *stream graphs* reste important. Pour mener à bien des applications concrètes, il est essentiel de concevoir et de mettre en œuvre des outils pour calculer efficacement les propriétés des *stream graphs*. Un pas dans cette direction a été effectué dans cette thèse par la conception d'algorithmes de connexité et de chemins ainsi que leurs implémentations dans Straph.

Comme mentionné, la théorie des *stream graphs* se situe à l'intersection de deux domaines scientifiques : la théorie des graphes et les séries temporelles. Dans le cadre de cette thèse, nous n'avons pas exploré les concepts de séries temporelles. Cependant, nous considérons que cet aspect est important et nous espérons fournir des avancées dans cette direction, notamment grâce à notre méthode permettant, pour différents types de propriétés dépendantes du temps, le calcul des séries temporelles associées.

Contents

Remerciements iii

Abstract v

Résumé vii

List of Figures xv

List of Tables xix

Introduction 1

1 Modeling of Temporal Networks: the Stream Graph Approach 5
- 1.1 Graph Theory . . . 5
- 1.2 Stream Graphs . . . 6
  - 1.2.1 Definitions and Notations . . . 8
  - 1.2.2 Related Works . . . 9
- 1.3 Real-World Stream Graphs . . . 12

2 Straph: A Python Library for Stream Graphs 15
- 2.1 Development Paradigms . . . 16
- 2.2 Data Structures . . . 17
  - 2.2.1 In-Memory Structures . . . 17
  - 2.2.2 Streaming Formats . . . 19
  - 2.2.3 File Formats . . . 20
- 2.3 Functionalities . . . 20
  - 2.3.1 Installation and Dependencies . . . 22
  - 2.3.2 Visualisation . . . 22
  - 2.3.3 Straph Generators . . . 24
- 2.4 Real-World Use Case: High School Friends . . . 31
- 2.5 Discussion: development choices and future features . . . 33

3 Connectivity 35
- 3.1 Weak Connectivity . . . 36
- 3.2 Strong Connectivity . . . 37
  - 3.2.1 Direct Approach . . . 39
  - 3.2.2 Fully Dynamic Approach . . . 41

3.2.3 Union-Find Approach . . . 42
3.3 Experiments and Applications . . . 44
3.3.1 Algorithm performances . . . 44
3.3.2 Connectedness analysis of IP traffic . . . 47
3.4 Related Work . . . 50
3.5 Conclusion . . . 51

4 Alternative Stream Graph Representations 53
4.1 Condensation . . . 54
4.1.1 Definitions . . . 55
4.1.2 Algorithm . . . 58
4.1.3 Connectivity Properties . . . 59
4.1.4 Reachability Queries . . . 62
4.1.5 Experiments . . . 63
4.2 Stable Directed Acyclic Graph . . . 66
4.2.1 Definitions . . . 66
4.2.2 Algorithm . . . 68
4.2.3 Experiments . . . 69
4.2.4 DAG Parallel Framework . . . 71
4.3 Δ-Approximation . . . 76
4.3.1 Approximate Strongly Connected Components . . . 76
4.3.2 Experiments . . . 77
4.3.3 Application to Latency Approximation . . . 78
4.4 Discussion . . . 80

5 Temporal Paths 81
5.1 Definitions . . . 82
5.2 Optimal Temporal Paths Problems . . . 82
5.2.1 Multi-criteria optimal temporal paths . . . 84
5.2.2 Dominated Paths . . . 87
5.3 L-Algorithm . . . 90
5.4 Condensation Based Algorithms . . . 98
5.4.1 Time to reach and foremost paths . . . 98
5.4.2 Latency and fastest paths . . . 104
5.5 Experiments . . . 106
5.6 Conclusion . . . 112

Conclusion 115

Bibliography 121

List of Figures

1-1 An example of stream graph. We display time $T = [0, 10]$ on the horizontal axis and nodes $V = \{A, B, C, D, E, F\}$ on the vertical one. We represent each node segment by a colored horizontal segment, with one color per node; and each link segment in grey by a vertical line between the two involved nodes at the link segment starting time, and a horizontal line from this time to its ending time. For instance, node A corresponds to two node segments: $([0, 5], A)$ and $([7, 10], A)$, meaning that $T_A = [0, 5] \cup [7, 10]$. There are two links segments between A and B: $([0, 4], AB)$ and $([7, 8], AB)$, meaning that $T_{AB} = [0, 4] \cup [7, 8]$. There is an instantaneous link segment: $([4, 4], BE) = (\{4\}, BE)$, and it is the only link between B and E, therefore $T_{BE} = [4, 4] = \{4\}$. . . 7

2-1 Diagram summing up Straph's functionalities 21

2-2 Straph drawings of a stream graph (left) and its aggregated graph (right) 23

2-3 Illustration of an animated drawing of a stream graph. The left view features a moving cursor over the time axis on the stream graph's global drawing, here at $t = 2.5$. On the right view, an induced graph is drawn corresponding to the cursor's position. 23

2-4 Illustrations of clustering visualisations in Straph. The instant degree value of each node (left) and clusters corresponding to the distinct strongly connected components of the stream graph (right). 24

2-5 Straph drawing of the temporal nodes degree (brighter the color higher the degree) in a subset of the High School dataset (substream of the first fifty nodes on the second day of recording). 25

2-6 An Erdo-Rényi generated stream graph (left) along with the visualisation of the degree of its temporal nodes (brighter the color higher the degree) 27

2-7 Degree distribution of a randomly generated Erdős-Rényi stream graph with parameters *T = (0, 1000), nb_nodes = 100000, occurrence_param_node = 4, presence_param_node = 200, occurrence_param_link = 3, presence_param_link = 150, p =* $\sqrt{nb\_nodes}/nb\_nodes$. 28

2-8 A Barabási-Albert generated stream graph (left) along with the visualisation of the degree of its temporal nodes (brighter the color higher the degree) 29

2-9 Degree distributions (top: histogram, bottom: log-log scatter plot) of a randomly generated Barabási-Albert stream graphs with parameters *T = (0, 1000), nb_nodes = 100000, occurrence_param_node = 3, presence_param_node = 500, occurrence_param_link = 3, presence_param_link = 250, m0 = 2, m= 2* 30
2-10 Substream induced by nodes *'275'*, *'312'*, *'612'*, *'886'* in the *High School* dataset. 32
2-11 Substream induced by nodes *'884'*, *'3'*,*'339'* and *'147'* in the *High School* dataset. 33

3-1 The two weakly connected components of the stream graph of Figure 1-1, each component having its own color. 36
3-2 The 17 strongly connected components of the stream graph of Figure 1-1. Each component is numbered and has its own color. 38
3-3 *SCC UF* algorithm illustration: The addition of a link ([2, 9],CD) between C and D causes component unions (U_0, U_1, U_2, U_3 and U_4). Components C_0 and C_1 are split at the begin of the link (instant 2) and component C_{10} remains unchanged. 42
3-4 Time cost of *SCC Direct*, *SCC UF*, *SCC FD* in seconds, along with the number M of link segments and the number of strongly connected components, for each considered stream (horizontal axis, ordered with respect to M). 46
3-5 Relation between the number of strongly connected components (horizontal axis) and $n*M$, the number of event times, and the running time of *SCC Direct*. Each dataset leads to three vertically aligned points, the color of which indicating the considered variable. 46
3-6 Number of nodes in the whole stream graph and in the giant weakly connected component over time in the *mawilab* dataset. 47
3-7 Number of nodes and number of strongly connected components over time in the *mawilab* dataset. 48
3-8 Distribution of the size (top left), duration (top right) and span (bottom) of strongly connected components in *Mawilab*. 48
3-9 Duration of each strongly connected component as a function of its size in *Mawilab* dataset, in log-log scales. We added 10^{-6} to each duration in order to display instantaneous SCC on a log scale. 49

4-1 The 17 strongly connected components of the stream graph of Figure 1-1 (top) and the corresponding Condensation Directed Acyclic Graph (bottom). 56
4-2 Illustration of the impact of a link's beginning on condensation links. 57
4-3 Relaxed Condensation Directed Acyclic Graph corresponding to the stream graph of Figure 1-1 59
4-4 A temporal path from $(0, F)$ to $(6, C)$ in the stream graph (top and middle) and the equivalent path in its condensation $((1, 2, 8, 4, 9, 5))$ (bottom). 61

4-5 Number of nodes in $G_{\mathscr{C}}$, n_c, along with the number of node segments in S, N, (left) and number of links in $G_{\mathscr{C}}$, m_c, along with the number of link segments in S, M, (right) for each considered real world stream graph (horizontal axis, ordered with respect to M). 64
4-6 Running time of *SCC-Condensation Direct* in seconds along with the number of link segments, M, node segments, N, and event times, T, in S for each considered real world stream graph (horizontal axis, ordered with respect to M). 65
4-7 Stable connected components of the stream graph of Figure 1-1. . . . 67
4-8 Illustration of the decomposition into snapshots. Colors indicate distinct snapshots and numbers indicate connected components in snapshots. 67
4-9 Number of nodes in $G_{\mathscr{S}}$, n_s along with the number of node segments in S, N (left) and number of links in $G_{\mathscr{S}}$, m_s along with the number of link segments in S, M (right) for each considered real world stream graph (horizontal axis, ordered with respect to M). 70
4-10 Running Time of *SCC-Condensation Direct* and *SCC-Stable Direct* in seconds along with the number of link segments M, node segments N and event times T in S for each considered real world stream graph (horizontal axis, ordered with respect to M). 70
4-11 Number of nodes in $G_{\mathscr{C}}$, n_c, and in $G_{\mathscr{S}}$, n_s (left) - Number of links in $G_{\mathscr{C}}$, m_c, and in $G_{\mathscr{S}}$, m_s (right) for each considered real world stream graph (horizontal axis, ordered with respect to M). 71
4-12 K-cores in a subset of the *Facebook* dataset (from the 01/07/2008 to the 22/01/2009). 74
4-13 Coreness distribution in the facebook dataset 75
4-14 Core Number of nodes '1055' (left) and '2420' (right) over time in the *facebook* dataset. 75
4-15 Running time of *SCC Direct*, number of SCC and number of event times in MawiLab, as a function of Δ (here, $\delta = 2s$). 77
4-16 Box plots representing the distribution of the size (left), duration (middle) and span (right) of strongly connected components in Mawilab, for various values of Δ (here, $\delta = 2s$). We indicate the mean, minimal, and maximal values with dots connected by horizontal lines, as well as the median and percentiles with vertical boxes. 78
4-17 Evolution of the LRMSE, the average difference between latencies and the average latency stretch with respect to Δ in Mawilab. We indicate the number of missing paths and represent it as a disk of area proportional to this number. 79

5-1 Examples of optimal temporal paths (top to bottom): *foremost path* from $(0, A)$ to F, *fastest path* from A to F, *shortest path* from A to D. 85
5-2 Examples of multi-criteria optimal temporal paths (top to bottom): *Shortest Foremost Path* $(0, A)$ to F, *Shortest Fastest Path* A to F, *Fastest Shortest Path* A to D. 86

5-3 Illustration of shortest fastest paths complexity 98
5-4 Running Times (s) of the *L-Algorithm* for the foremost, shortest fastest, shortest foremost, fastest shortest and shortest fastest path problems (top: random sources, bottom: high degree sources). 109
5-5 Running Times (s) of the *L-Algorithm* and of condensation algorithms for the foremost and fastest path problems (top: random sources, bottom: high degree sources). 110

List of Tables

1.1 Notations 10

1.2 Key features of the real-world stream graphs we consider, ordered with respect to their number M of link segments (K indicates thousands, M millions). n the number of distinct nodes, m the number of distinct links, $|T|$ the considered time window, N the number of node segments, M the number of link segments, Ω the number of distinct event times and d_m the maximal instantaneous degree. 14

3.1 Number of WCC ($|\mathscr{W}|$) - Number of SCC ($|\mathscr{C}|$) - Algorithms running time in seconds (K = 10^3, M=10^6) 45

4.1 Running time in seconds of algorithm *SCC-Condensation Direct* - Characteristics of the condensations of real world stream graphs ($n_c = |\mathscr{C}|$) the number of nodes (and of SCC), m_c the number of links, d_c the mean out degree) - $\alpha = \frac{\sum_{u \in \mathscr{C}} d_{out}(u)^2}{n_c}$ the complexity parameter in Heuristic 4.1.1 - $|\mathscr{W}|$ the number of weakly connected components. . . 64

4.2 Running time in seconds of algorithm *SCC-Stable Direct* - Characteristics of the stable DAG of real world stream graphs (n_s the number of nodes, m_s the number of links,d_s the mean out-degree) - Total number of connected components in the whole sequence of snapshots 69

4.3 Characteristics of the 3-core, 2-shell and 1-shell in the *Facebook* dataset. The total running time of the DAG parallel framework, with 8 cores, was 32.04s. 73

5.1 Example of *L-Algorithm* for shortest paths ($\mathcal{F}_{SP}$ and $\mathcal{Q}_{SP}$). 93

5.2 L-Algorithm running times in seconds (random sources) (missing values correspond to the timeout of a procedure, set to 15000s) 108

5.3 L-Algorithm running times in seconds (high degree sources) (missing values correspond to the timeout of a procedure, set to 15000s) . . . 111

5.4 L-Algorithm and condensation algorithms running times in seconds (random sources) (missing values correspond to the timeout of a procedure, set to 15000s) 111

5.5 L-Algorithm and condensation algorithms running times in seconds (high degree sources) (missing values correspond to the timeout of a procedure, set to 15000s 112

Introduction

*A **network** is composed of actors and connections between these actors. Actors are called **nodes** and connections **links**. The term **network** refers to real-world complex systems where nodes are complex entities. The term **graph** refers to the abstract mathematical object modeling relations between these entities.*

Networks are ubiquitous in our modern societies. Every aspect of our social life is a network, whether it is a social network, a messaging or dating app, a purchase or a sale, or even physical proximity between individuals. Relationship networks are the most obvious domain where these data shine, as they provide meaningful information to model and understand social interactions.
Analyzing academic collaborations, e-mail exchanges and, more generally, social networks is a major concern. For instance, the increase and spread of fake news has a growing and worrying influence on public opinion. Data collection is accelerating due to technological advances in recent years. The entailed risks and the wide range of applications from marketing, to military, to health care of these data require a better understanding of networks.
Likewise, the analysis of infrastructure networks such as road or telecommunication networks, has a plethora of applications. Even data that does not inherently contain any network structure can be analyzed from a network science perspective. Consider, for example, text data: a network of terms can be built by assigning links between adjacent words (terms). Many scientific problems can be reduced to a specific graph problem in a similar way. DNA sequencing from small DNA fragments consists in finding a topological sort of a directed graph built from these fragments. Graph modeling of biological data, from DNA sequencing to molecule similarity, has already numerous applications. Such abstract representations of real-world problems have proven to be decisive in our understanding of many phenomena. Furthermore, from a practical point of view, graph algorithms have demonstrated their efficiency for solving or approximating many problems. For instance, finding the cheapest route from a town to another one consists in solving a classical graph problem: the shortest path problem in a weighted graph.

Network science is a long-existing field of research and has already tackled efficiently many of the above problems, providing algorithms and tools for the analysis and modeling of networks, see for instance [14, 19, 104, 106, 72, 9]. However, network

science has, for a long time, been limited to static phenomena. More precisely, it has been applied to networks without taking into account the intrinsic temporal nature of many phenomena. However, if we focus on relations between individuals it is impossible to ignore the temporal nature of human interactions without losing information. The approach guiding our work is to bring this temporal dimension into the modeling in the form of networks. The formalism we use in this thesis is the *stream graph theory* [59]. Similarly to time series theory providing mathematical tools and a specific modeling to analyze signals as continuous sequences of data points, stream graph theory provides tools for the analysis and modeling of sequences of links or, continuous sequences of graphs. In addition, this formalism is consistent with graph theory: if a stream has no dynamics, it is equivalent to a classical graph and its properties are same as those of this graph [59].

Thesis Aims and Objectives

The goal of this thesis is twofold: it is to take benefit from the stream graph formalism in order to analyze the dynamics and structure of real-world data; and in return to benefit from these practical applications to extend and improve the formalism, in particular to solve the algorithmic challenges associated with its implementation on massive data.

Numerous stream graph notions, such as connected components or paths, were defined recently in [59], but no algorithm was provided to compute them. Consequently, no experimental evaluation of these concepts have been performed and whether these concepts are suitable for describing real-world temporal networks was an open question.
In this thesis we aim to provide efficient algorithms to compute these concepts, analyze their algorithmic complexities and their practical performances. These implementations and the design of appropriate data structure(s) for the efficient handling of stream graphs will provide an advanced tool in the form of a python library: *Straph*.
In addition to these algorithmic issues, we will endeavour to describe real-world stream graphs with these concepts and evaluate how they help us understanding the structure and dynamics of these complex objects. We also aim at identifying the most relevant notions introduced, in the sense that they are calculable in practice and bring significant insight into the data. This will allow us to evaluate the added value of the stream graph approach and to explore its limits.

Main objectives and Challenges

- Solve algorithmic questions related to the computation of connectivity and path concepts in stream graphs, especially on massive datasets.
- Develop a python library to analyze stream graphs, using parallel and/or streaming implementations in order to scale on real-world datasets.
- Use the concepts defined in the formalism to analyze the structure and dynamics of real-world large-scale data.

Thesis Outline and Contributions

In Chapter 1 we present the stream graph formalism and detail why it is particularly conducive to model highly dynamic networks in which nodes and/or links arrive and/or leave over time.

Then we present, in Chapter 2, the software that we have developed during this thesis: the python library *Straph*. This tool was specifically designed to handle stream graphs. We will explain the motivation for such a tool, overview its core functionalities and provide practical examples throughout this thesis. Along with this tool, we introduce several data structures suited to handle different stream graph concepts as well as random stream graph generators based on the Erdős-Rényi and Barabási-Albert models.

In Chapter 3 we focus on connectivity concepts in stream graphs. We propose several algorithms, with polynomial time and space complexities, to compute connectivity concepts. We provide an implementation and experimentally compare the algorithms in a wide variety of practical cases. We then conduct the first connectivity analysis of a representative large-scale dataset and show that connected components indeed give much insight on its features. In particular, we show and explain that real-world stream graphs generally do not have a huge strongly connected component that stands out of the crowd.

In order to answer some issues raised in Chapter 2 about the data structures used to handle stream graphs, we propose, in Chapter 4, two alternative data structures: the condensation and the stable directed acyclic graphs. These objects benefits from the connectivity notions and algorithms previously defined in Chapter 3. We show that these objects can be used to compute reachability queries in linear time and provide a parallel framework to efficiently compute numerous stream graph properties using graph algorithms. In addition, we propose an approximation scheme that significantly reduces computation costs, and gives even more insight on the dataset.

Finally, Chapter 5 is dedicated to temporal path problems arising in stream graphs. After presenting a theoretical overview of the different kind of algorithmic path problems, we propose a new generic algorithm able to compute optimal temporal paths in polynomial time. In a second time, using results from Chapters 3 and 4, we will pro-

vide the first linear algorithms solving two optimal path problems in stream graphs. We conclude this thesis by discussing the practical usage and improvement of our work as well as the impact it could have in many scientific fields.

1 Modeling of Temporal Networks: the Stream Graph Approach

Many real world complex networks have a temporal dimension, such as contacts between individuals, financial transactions or network traffic. Graph theory provides a wide set of tools to model and analyze static connections between entities. Unfortunately, this approach does not take into account the temporal nature of interactions. One fallback consists in decomposing these interactions into a sequence of static graphs (by aggregation over a period of time or by adding temporal attributes on edges and nodes). However, this solution either leads to a reduction of information or to a potentially huge sequence of graphs (see section 1.2.2). In the frame of our work, we use the *stream graph* formalism [59] to directly cope with interactions over time without loss of information.

First we present common and relevant graph concepts as well as their key properties, in section 1.1. Then we present equivalent concepts for stream graphs, that were defined in [59], along with their key properties in section 1.2. We will provide the basic notations, definitions and material that will be used through this thesis. We will also review different approaches designed to model temporal networks and point out their similarities and differences with ours. Finally, in section 1.3 we show how the stream graph is particularly well suited to model highly dynamic structural data. We also describe and present key properties of some representative real world datasets of different scales.

1.1 Graph Theory

Given any two sets A and B, we denote by $A \times B$ the set of couples (a, b) such that $a \in A$ and $b \in B$. We denote by $A \otimes B$ the set of pairs ab such that $a \in A$, $b \in B$ and $a \neq b$. Couples are ordered, while pairs are unordered: $(a, b) \neq (b, a)$ while $ab = ba$.

Definition 1.1.1. *A graph is a pair $G = (V, E)$, where V is a set whose elements are called nodes and $E \subseteq V \otimes V$ a set of two-sets (sets with two distinct elements) whose elements are called edges.*

A graph is a representation of entities linked by some relationships. These relationships can differ a lot depending on the application domain (see section 1.3). The number of nodes in a graph is equal to $n = |V|$ and the number of edges is $m = |E|$. Also a graph can be directed, undirected, bipartite, multipartite, can have weighted or unweighted edges, and nodes can have labels or attributes.

Definition 1.1.2. *A subgraph of a graph $G = (V, E)$ is formed from a subset of nodes, C, and from all the edges that have both endpoints in the subset. A subgraph induced by a set $C \subseteq V$ is denoted by $G(C) = (C, E_C)$.*

Definition 1.1.3. *A path P from $u \in V$ to $v \in V$ of a Graph $G = (V, E)$ is a sequence $(u_0, v_0), (u_1, v_1), ..., (u_k, v_k)$ of elements of $V \times V$ such that $u_0 = u$, $v_k = v$, and for all i, $u_i = v_{i-1}$ and $(u_i, v_i) \in E$. The length of the path $P = (u_0, v_0), (u_1, v_1), ..., (u_k, v_k)$ is the number of its elements : $k + 1$. The path P is a shortest path from u to v if there is no path from u to v in G of length lower than k. The distance between u and v is the length of the shortest path between them.*

Definition 1.1.4. *A connected component of a graph $G = (V, E)$ is a maximal set $C \subseteq V$ such that $\forall u, v \in C$ there exists a path from u to v in the induced subgraph $G(C)$.*

A disconnected graph is disconnected if there exists two nodes $u, v \in V$ such that there is no path from u to v in G. The nodes set of such a graph can be partitioned into a set of distinct connected components.

Definition 1.1.5. *The k-core of a graph $G = (V, E)$ is its largest subset $C^k \subseteq V$ such that $\forall v \in C^k, d(v) \geq k$ in the induced subgraph $G(C^k) = (C^k, E_{C^k})$.*

Definition 1.1.6. *The k-shell of a graph $G = (V, E)$ is the subset $C^k \setminus C^{k+1}$. (defined only if the (k+1)-core is not empty : $C^{k+1} \neq \emptyset$).*

Definition 1.1.7. *A clique of a graph $G = (V, E)$ is a subset $C \in V$ such that all pairs of nodes involved in C are linked together in G. A clique involving k nodes is called a k-clique. A clique C is maximal if there is no other clique C' such that $C \subset C'$.*

Graph theory is a long-existing research field. Many graph concepts have been defined as well as algorithms to compute them. These concepts have led to numerous real-world applications. We refer to [14, 19, 104, 106, 72, 9] for a detailed survey of graph theory notions and their practical applications.

1.2 Stream Graphs

Definition 1.2.1. *A **stream graph** $S = (T, V, W, E)$ is defined [59] by a finite set of nodes V, a time interval $T \subseteq \mathbb{R}$, a set of temporal nodes $W \subseteq T \times V$, and a set of links $E \subseteq T \times V \otimes V$ such that $(t, uv) \in E$ implies $(t, u) \in W$ and $(t, v) \in W$.*

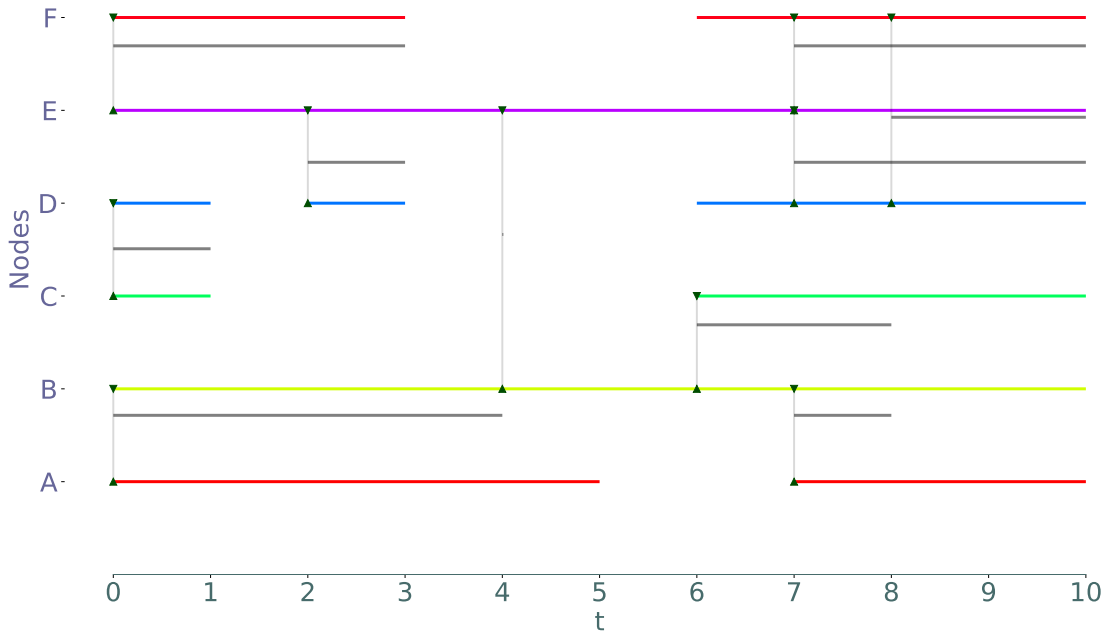

Figure 1-1: An example of stream graph. We display time $T = [0, 10]$ on the horizontal axis and nodes $V = \{A, B, C, D, E, F\}$ on the vertical one. We represent each node segment by a colored horizontal segment, with one color per node; and each link segment in grey by a vertical line between the two involved nodes at the link segment starting time, and a horizontal line from this time to its ending time. For instance, node A corresponds to two node segments: $([0, 5], A)$ and $([7, 10], A)$, meaning that $T_A = [0, 5] \cup [7, 10]$. There are two links segments between A and B: $([0, 4], AB)$ and $([7, 8], AB)$, meaning that $T_{AB} = [0, 4] \cup [7, 8]$. There is an instantaneous link segment: $([4, 4], BE) = (\{4\}, BE)$, and it is the only link between B and E, therefore $T_{BE} = [4, 4] = \{4\}$.

Figure 1-1 shows an example of a Stream Graph. In this example, nodes may represent individuals, their presence reflects the fact that they are in the same room and interactions between individuals could represent a conversation. The node C is present from 0 to 1, is absent between 1 and 6, and present again from 6 to 10. Nodes A and B are continuously connected from 0 to 4 and from 7 to 8. Interactions can be instantaneous like between B and E at time 4.

1.2.1 Definitions and Notations

The following stream graph properties are **consistent**: if one considers a stream graph with no dynamics - nodes are present all the time, and two nodes are either linked all the time or not at all - then the stream graph is equivalent to a graph and its stream properties are equivalent to the properties of the corresponding graph [59].

Definition 1.2.2. *In a stream graph $S = (T, V, W, E)$ a path P from $(\alpha, u) \in W$ to $(\omega, v) \in W$ is a sequence $(t_0, u_0, v_0), (t_1, u_1, v_1), ..., (t_k, u_k, v_k)$ of elements of $T \times V \times V$ such that $u_0 = u$, $v_k = v$, $t_0 \leq \alpha$, $t_k \geq \omega$ for all i, $t_i \leq t_{i+1}$, $v_i = u_{i+1}$ and $(t_i, u_i v_i) \in E$, $[\alpha, t_0] \times u \subseteq W$, $[t_k, \omega] \times v \subseteq W$, and for all i, $[t_i, t_{i+1}] \times v_i \subseteq W$.*

Definition 1.2.3. *A path $P = (t_0, u_0, v_0), (t_1, u_1, v_1), ..., (t_k, u_k, v_k)$ has length $k + 1$ and duration $t_k - t_0$.*

Definition 1.2.4. *A stream $S' = (T', V', W', E')$ is a **substream** of a stream graph $S = (T, V, W, E)$ if $T' \subseteq T$, $V' \subseteq V$, $W' \subseteq W$ and $E' \subseteq E$. It is denoted by $S' \subseteq S$.*

Definition 1.2.5. *A **cluster** C of a Stream Graph $S = (T, V, W, E)$ is a subset of W.*
*The set of links between nodes involved in C is denoted by $E(C) = \{(t, uv) \in E, (t, u) \in C$ and $(t, v) \in C\}$, the **substream of** S **induced by** C is denoted by $S(C) = (T, V, C, E(C))$.*

Definition 1.2.6. *A weakly connected component of a Stream Graph $S = (T, V, W, E)$ is a maximal cluster $C \subseteq W$ where $\forall (\alpha, u), (\omega, v) \in W_C^2$, (α, u) - - - (ω, v) (There exists an undirected path between (α, u) and (ω, v)) in the induced sub-stream $S(C) = (T_C, V_C, W_C, E_C)$.*

Definition 1.2.7. *A strongly connected component of a Stream Graph $S = (T, V, W, E)$ is a maximal compact cluster $C \subseteq T \times V$ such that V_C is a connected component of G_t, the induced static graph of S at time t, for all t in T_C. This implies that $\forall u, v \in V_C^2$ and $\forall t \in T_C$ there exists a path between u and v in G_t.*

Definition 1.2.8. *The k-core of a stream graph $S = (T, V, W, E)$ is its largest cluster $C^k \subseteq W$ such that $\forall (t, v) \in C^k, d_t(v) \geq k$ in the induced sub-stream $S(C^k)$.*

Definition 1.2.9. *The k-shell of a stream graph is the cluster $C^k \setminus C^{k+1}$ (the k-shell is defined only if the (k+1)-core is not empty : $C^{k+1} \neq \emptyset$).*

Definition 1.2.10. *A clique of stream graph S is a cluster C of S of density 1. In other words, all pairs of nodes involved in C are linked in S whenever both are*

involved in C. A clique C is maximal if there is no other clique C' such that $C \subset C'$. A clique involving k distinct nodes is called a k-clique.

For the sake of clarity, we do not present an exhaustive list of stream graphs definitions. We refer to [59] for a detailed presentation of stream graph theory. Additional definitions will be recalled in appropriated sections of this thesis.

For any u and v in V, $T_u = \{t, (t, u) \in W\}$ denotes the set of time instants at which u is present, and $T_{uv} = \{t, (t, uv) \in E\}$ denotes the set of time instants at which u and v are linked together. We assume here that both T_u and T_{uv} are unions of a finite number of disjoint closed intervals (possibly singletons) of T.

We call *node segment* a couple $([b, e], u)$ such that $[b, e]$ is a maximal interval in T_u, and we denote by $\overline{W}$ the set of all node segments in W. We say that b is an *arrival* of u, and e a *departure*. We denote by $N = |\overline{W}|$ the number of node segments in the stream.
Likewise we call *link segment* a couple $([b, e], uv)$ such that $[b, e]$ is a maximal interval in T_{uv}, and we denote by $\overline{E}$ the set of all link segments in E. We say that b is an *arrival* of uv, and e a *departure*. We denote by $M = |\overline{E}|$ the number of link segments in the stream.
Notice that the intervals considered above may be singletons. Then, $b = e$ and $[b, e] = \{b\} = \{e\}$.

We call all time instants that correspond to a node or link arrival or departure an *event time*. The number of distinct event times is denoted by Ω. There are at most $2 \cdot N + 2 \cdot M$ event times in a stream graph.

The induced graph $G(S) = (V(S), E(S))$ is defined by $V(S) = \{v, \exists t, (t, v) \in W\}$ and $E(S) = \{uv, \exists t, (t, uv) \in E\}$. We denote by $n = |V(S)|$ and $m = |E(S)|$ its number of nodes and links, respectively.

We denote by $G_t = (V_t, E_t)$ the graph such that $V_t = \{v, (t, v) \in W\}$ and $E_t = \{uv, (t, uv) \in E\}$. We denote by G_t^- the graph that corresponds to the nodes and links present between t and the event time just before it: $G_t^- = (V_t^-, E_t^-)$ where $V_t^- = \{v, \exists t' \neq t, [t', t] \subseteq T_v\}$ and $E_t^- = \{uv, \exists t' \neq t, [t', t] \subseteq T_{uv}\}$.

In this thesis we will use the notations of Table 1.1 to ease the explanation of several methods and algorithms.

1.2.2 Related Works

Sequences of interactions are preponderant in numerous fields, and they have been studied for a long time. Although many variations exist, the most common approach is to model them by sequences of graphs (each graph aggregating interactions occurring during a period of time), by labeled graphs (each link being labeled with its presence times), or other augmented graphs. This makes it possible to use graph theory to study these sequences of graphs, labeled graphs, and other variants. Other works deal directly with higher-level methods for studying graphs, like stochastic block models

Notation	Definition	Explanation
$V(S)$	$\{v, \exists t, (t, v) \in W\}$	Set of distinct nodes
W	$W \subseteq T \times V$	Set of temporal nodes
$\overline{W}$	$\{([b, e], v)$ s.t. $[b, e] \times \{v\} \subseteq W$ and $[b, e]$ is maximal$\}$	Set of node segments
W^*	$\{([b, e], v), (e, v)$ s.t. $\exists([b, e], v) \in \overline{W}\}$	Temporally ordered set of node events
E	$E \subseteq T \times (V \otimes V)$	Set of temporal links
$\hat{E}$	$\{uv$ s.t. $\exists t, (t, uv) \in E\}$	Set of distinct links
$\overline{E}$	$\{([b, e], uv) s.t. [b, e] \times \{uv\} \subseteq E$ and $[b, e]$ is maximal$\}$	Set of link segments
$E(S)$	$\{([b, e], uv)$, (e, uv) s.t. $\exists([b, e], uv) \in \overline{E}\}$	Temporally ordered set of link events
Π	$W^* \cup E^*$	Temporally ordered set of events
T_u	$\{t, (t, u) \in W\} \subseteq T$	Set of time instants at which u is present
$\overline{T_u}$	$\{[b, e]$ s.t. $\exists([b, e], u) \in \overline{W}$ and $[b, e]$ is maximal $\}$	Set of maximal distinct intervals of presence of u
T_u^*	$\{b, e$ s.t. $\exists[b, e] \in \overline{T_u}\}$	Set of time instants at which u arrives or departs
T_{uv}	$\{t, (t, uv) \in E\} \subseteq T$	Set of time instants at which uv is present
$\overline{T_{uv}}$	$\{[b, e]$ s.t. $\exists([b, e], uv) \in \overline{E}$ and $[b, e]$ is maximal $\}$	Set of maximal distinct intervals of presence of uv
T_{uv}^*	$\{b, e$ s.t. $\exists[b, e] \in \overline{T_{uv}}\}$	Set of time instants where uv begins or ends
Ω	$\lvert\Pi\rvert$	Set of event times
n	$\lvert V\rvert$	Number of distinct nodes
m	$\lvert\hat{E}\rvert$	Number of distinct links
N	$\lvert\overline{W}\rvert$	Number of distinct node segments
M	$\lvert\overline{E}\rvert$	Number of distinct link segments

Table 1.1: Notations

for instance, and extend them to cope with the dynamics. Finally, a few works define specific properties combining temporal and structural information. We detail these various works below.

Studying interactions over time is crucial in a wide variety of contexts, such as studies of phone call [56, 17], contacts between individuals [10, 65, 101], messaging [39, 35], or internet traffic [45, 102]. In each practical context, researchers face the challenge of analyzing the both temporal and structural nature of interactions, and they develop ad hoc methods and tools to do so. Several surveys of these works are available from various perspectives [58, 90, 97, 91, 47, 38, 28]. The most classical approach consists in splitting time into slices and then building a graph, called *snapshot*, for each time slice: its nodes and links represent the interactions that occurred during this time slice. One obtains a sequence of snapshots (one for each slice), and may study the time evolution of their properties, see for instance [88, 61, 83, 41, 18, 99], among many others. In [12], the authors even design a general framework to combine and aggregate wide classes of temporal properties, thus providing a unified approach for snapshot sequence studies.

However, these approaches need time slices large enough to ensure that each snapshot captures significant information. But large slices lead to losses of temporal information, since all interactions within a same slice are merged. In addition, several or even varying slice durations may be relevant. As a consequence, choosing appropriate time slices is a research topic in itself [64, 80, 57]. To avoid these issues, several authors propose to encode the full information into various kinds of augmented graphs. In [22, 12] for instance, authors consider the graph of all nodes and links occurring within the data, and label each node and link with its presence times. In [105, 55, 68], the authors duplicate each node into as many copies as its number of occurrences (they assume discrete time steps); then, an interaction between two nodes at a given time is encoded by a link between the copies of these nodes at this time, and each copy of a node is connected to its copy at the next time step. All these approaches have a clear advantage: once the data is transformed into one or several graphs, it is possible to use graph tools and concepts to study the interactions under concern.

Various powerful methods for graph studies are extended to cope with the dynamics. This leads for instance to algebraic approaches for temporal network analysis [12], dynamic stochastic block models [111, 67, 25], dynamic Markovian models [92, 91], signals on temporal networks [44] or adjacency tensors [94, 37]. These works extend higher-level methods to the temporal setting. Complementary to these approaches that extend methods, some works extend various graph concepts to deal with time [12, 47, 74], similarly to the stream graph approach.

Whereas there is a very rich body of works and theoretical modeling of temporal networks (time-varying graphs, longitudinal networks, stochastic block models, markovian models, ...) none of these works aims at extending basic graph theory concepts where time and structure are equally important which is the main focus of the stream graph approach.

1.3 Real-World Stream Graphs

Stream graphs are particularly suited to model temporal networks with a highly internal dynamics. In order to explore the performances of our algorithms in a wide variety of situations, we considered 14 publicly available datasets that we shortly present below.
First notice that most available datasets record instantaneous interactions only, either because of periodic measurements, or because only one timestamp is available. In such situations, one resorts to δ-analysis [59]: one considers that each interaction lasts for a given duration δ. More precisely, if a dataset consists in a set D of triplets (t, uv) indicating a link between u and v at time t, then we consider $E = \cup_{(t,uv)\in D}[t, t+\delta] \times \{uv\}$. We also define V as the set of nodes that occur in D, T as the smallest interval of $\mathbb{R}$ that contains all times occurring in E, and W as the set of all (t, v) in $T \times V$ such that $\exists u, (t, uv) \in E$. This transforms D into a stream graph $S = (T, V, W, E)$ in which all link segments last for at least δ, and all links in D separated by a delay lower than δ lead to a unique link segment. Nodes are considered as present only when they have at least one link.

The key stream graph properties of the 14 considered datasets are given in Table 1.2, together with the value of δ we used. It either corresponds to a natural value underlying the dataset or is determined by the original timestamp precision.

UC Message (UC) [75]: is a capture of messages between University of California students in an online community. A node represents a user. An edge represents a sent message.

High School 2012 (HS 2012) [33]: is a recording of interactions between students of 5 classes during 7 days in a high school in Marseille, France in 2012. An interaction consists in a physical proximity between two students, captured by a sensor.

Digg [24, 2]: is the reply network of the social news website Digg. Each node in the network is a user of the website, and each edge denotes that a user replied to another user.

Infectious [50]: is a recording of face-to-face contacts between visitors of the INFECTIOUS: STAY AWAY exhibition in 2009, Dublin. Nodes represent exhibition visitors; edges represent face-to-face contacts that were active for at least 20 seconds.

Twitter Higs (Twitter) [27, 62]: is a recording of user activities in Twitter (retweeting, replying to existing tweets, mentioning other users, friends/followers social relationships among users involved in the above activities) for one week around the discovery of the Higgs boson in 2012.

Linux Kernel mailing list (Linux) [4]: represents the email replies between users on the Linux kernel mailing list. Nodes are users (identified by their email addresses), and each edge represents a reply from a user to another.

Facebook wall posts (Facebook) [103] : Messages exchanged between users on

Facebook. The nodes of the network are Facebook users, and each edge represents one post, linking the user writing a post to the user whose wall the post is written on.

Epinions [3, 66]: is the trust and distrust network of Epinions, an online product rating site. The network consists of individual users connected by trust and distrust links.

Amazon [1, 63]: is a bipartite network containing product ratings from the Amazon online shopping website. Nodes represent users and products, and edges represent individual ratings.

Youtube [69]: is a social network of YouTube users and their friendship connections.

Movielens [40]: This bipartite network contains ten million movie ratings from Movielens. Left nodes are users and right nodes are movies. An edge between a user and a movie indicates that the user has rated the movie.

Wiki Talk En (Wiki) [95]: is a recording of discussions between contributors to the English Wikipedia. Nodes represent users of the English Wikipedia, and an edge from user A to user B denotes that user A wrote a message on the talk page of user B at a certain timestamp.

Mawilab 2020-03-09 (Mawilab) [32]: is a 15 minutes capture of network traffic on a backbone trans-pacific router in Japan on March 3, 2020. Each link represents a packet exchanged between two internet addresses.

Stackoverflow [77, 62]: is a recording of interactions on the stack overflow website.

We have chosen a panel of datasets that we consider to be representative of publicly available datasets. They represent several real-world phenomena, providing different types of interactions at various frequencies. These interactions can represent email exchanges (*Enron*), conversations and following relationship on social networks (*Twitter*, *Facebook*, *Youtube*, *Digg*), discussions on online forums (*Wikipedia*, *Stackoverflow*, *Linux*, *UC Message*), ratings of products (*Movielens*, *Amazon*), network traffic (*Mawilab*) as well as human contacts (*High School*, *Infectious*).

As presented in Table 1.2, these datasets are of various sizes, span different periods of time, have repeated interactions ($m << M$) or not ($m \sim M$), have different connectivity profiles (d_m the maximal instantaneous degree varying from 11 to 28 710) and have various number of distinct event times (Ω ranging from 205 to 56 millions).

	δ	n	m	$\lvert T\rvert$	N	M	Ω	d_m
UC	1h	2K	14K	189d	43K	34K	67K	98
HS 2012	60s	327	6K	4d	48K	46K	7K	11
Digg	1h	30K	85K	14.5d	110K	86K	158K	19
Infectious	60s	11K	45K	80d	85K	133K	63K	12
Twitter	600s	304K	452K	7d	543K	488K	271K	1 435
Linux	10h	27K	160K	8y	450K	544K	913K	67
Facebook	10h	46K	183K	4.3y	957K	588K	1.2M	56
Epinions	10h	132K	711K	2.6y	404K	743K	2K	2 827
Amazon	1h	2.1M	5.7M	9.5y	9.9M	5.8M	7K	272
Youtube	24h	3.2M	9.4M	226d	6.7M	9.4M	205	28 710
Movielens	1h	70K	10M	14y	8.5M	10M	14M	1 426
Wiki	1h	2.9M	8.1M	14.3y	18.3M	14.5M	27M	28 710
Mawilab	2s	940K	9.1M	902s	17M	18.8M	35.1M	16 384
Stackoverflow	10h	2.6M	28.2M	7.6y	30M	33.5M	56M	110

Table 1.2: Key features of the real-world stream graphs we consider, ordered with respect to their number M of link segments (K indicates thousands, M millions). n the number of distinct nodes, m the number of distinct links, $|T|$ the considered time window, N the number of node segments, M the number of link segments, Ω the number of distinct event times and d_m the maximal instantaneous degree.

2 Straph: A Python Library for Stream Graphs

Contributions

- The most advanced open source python library for the manipulation, modeling, analysis and visualisation of stream graphs: Straph
- Efficient data structures to handle stream graphs
- Two random stream graph generators based on the Erdős-Rényi and Barabási-Albert models

The number of applications of stream graph theory has risen rapidly, as shown in chapter 1, along with the number of theoretical concepts and algorithms to compute them. These needs motivate the development of an advanced tool to manipulate, analyse and visualise stream graphs. Straph [79] was developed in order to have a reliable library for handling stream graphs, to design algorithms and models, and to rapidly evaluate them.

At the time of development, alternative libraries were unsatisfactory. Either these libraries did not model time as continuous or they did not scale up on large datasets. The main challenge, in order to handle huge temporal networks, resides in choosing the right data structure to manipulate these kinds of networks. Focusing on graph libraries, we can distinguish two approaches: in NetworkX [43] nodes are modeled by python objects and in NetworKit [93] by integers. Consequently, links are represented by a pair of references to python object or by a pair of integers. This design choice is critical: the first option leads to a flexible framework where nodes can be as complex as one desires, the second option allows more compact representations.

Many libraries handling temporal networks focus on information diffusion and epidemic spreading. A comparison of these libraries can be found in [82]. The most complete one, DyNetx [82], uses a NetworkX graph object as a basic structure and overloads it to handle the temporal dimension. Hence, this library has the same flaw as NetworkX: it cannot handle huge datasets.
Two other libraries aim at modelling temporal networks: Pathpy [84] and Teneto [98]. However, in both of them, time is modeled as discrete therefore they cannot be used

in the stream graph case.
Stream_graph [87] is a library specifically designed to handle stream graphs. It uses Numpy arrays and Pandas dataframes to represent nodes and links as well as their intervals of presence. As we will show in section 2.2 this choice may be more user-friendly but degrades overall performances in practice.

Straph is an open source Python 3 package, under the licence Apache 2.0, for exploration and analysis of real and artificial stream graphs. This library provides specific data structures for representing different types of stream graphs, algorithms to compute basic properties and measures, readers and writers for various data formats as well as generators similar to Erdős-Rényi [30] and Barabási-Albert [7] models. In the long term, we are hoping to provide the equivalent of Networkx or Networkit, in terms of functionalities, for stream graphs.
Straph can be used to teach stream graph theory or illustrate particular concepts. Several Jupyter notebooks have been written in order to demonstrate its functionalities. Straph can be used by users or developers that are not necessarily experts in programming or in stream graph theory.

In this chapter we will detail the paradigms behind the development of Straph (Section 2.1) and we will overview the architecture of the library as well as the employed data structures (Section 2.2). Then we present the many features and possibilities of Straph (Section 2.3) and illustrate how to use them in practice (Section 2.4). Finally, we will discuss potential and future features (Section 2.5).

2.1 Development Paradigms

The choice of the programming language, Python, was motivated by its popularity, flexibility and compatibility with other existing frameworks. Moreover, Python is easy to read, write and use, hence, facilitates community contribution. Straph is based on several paradigms, each one having its advantages and drawbacks:

- The fewer dependencies the better: in order to facilitate its usage and to allow a better compatibility over time.
- Efficient and compact data structures: as the main objective is to handle stream graphs with millions of interactions the basic data structures of Straph should be as compact as possible while preserving a low querying complexity.
- Comprehensive and user-friendly API: we avoid framework code and keep the number of different objects to a minimum. We also provide an API similar to NetworkX to facilitate its usage by the graph community.
- Support growth and improvement: as an open source tool it must be easy for developers to contribute to functionalities of Straph.

2.2 Data Structures

The data structures used as basic blocks in a computational library are critical from a performance and memory consumption point of view. In the following we detail the data structures used in Straph as well as supported file and data formats.

2.2.1 In-Memory Structures

A stream graph can be composed of tens of millions of links and nodes. Each link and node having its own event times, this leads to the construction of numerous structures. For the sake of efficiency and simplicity, in Straph, we choose to limit basic data structure to built-in python objects. Temporal nodes or links could have been stored in more efficient data structures such as Numpy arrays, but the construction of these objects is time consuming and adds a layer of complexity.

Stream Graph **Object**

A stream graph being a complex structure evolving through time, several data structures are necessary to computationally represent one. The topological structure is encoded in two arrays *nodes* and *links*. The dynamics is encoded in two arrays of arrays *node presence* and *link presence*. The time window of a stream graph, T, is encoded by its bounds, a couple: *times*.

For efficiency reasons, nodes in Straph are encoded by integers typically from 0 to $n-1$, n being the number of distinct nodes ($|V|$) in the stream graph. However it is possible to assign any label, a string or a number and even an object to each node using a python dictionary. Consequently, a link in Straph is encoded by a couple of integers.

Node dynamics is encoded in the array of arrays *node presence*. Each array corresponds to a node's event times. Elements with an even index correspond to an arrival time and elements with an odd index to a departure time. Link dynamics is represented similarly in *link presence*.

The space complexity of this representation is in $\Theta(M+N)$, M and N being the number of distinct link and node segments (see chapter 1). A node and its sequence of event times are accessible in $O(1)$ while a link and its sequence of event times are accessible in $O(M)$.

For instance, a stream graph $S = (T, V, W, E)$ where $T = [0, 10]$, $V = \{A, B\}$, $W = ([0, 5] \cup [6, 10]) \times A \cup [1, 10] \times B$, $E = ([1, 4] \cup [6, 7]) \times (A, B)$ will be encoded by:

```
nodes = [0,1]
links = [(0,1)]
node_to_label = {0:'A', 1:'B'}
node_presence = [[0,5,6,10],[1,10]]
link_presence = [[1,4,6,7]]
S = stream_graph(times = (0,10),
                 nodes = nodes,
                 links = links,
                 node_to_label = node_to_label,
                 node_presence = node_presence,
                 link_presence = link_presence
                 )
```

This compact representation allows us to manipulate, in-memory, stream graphs with more than one hundred millions of temporal links, such as *Mawilab* datasets or *Stackoverflow*, on a modern Laptop computer.

Another compact encoding of a stream graph consists in the above structures *node* and *node presence* associated to a temporal adjacency list in which each node is linked to its neighbors and their corresponding times of interactions. For instance, S, as defined previously, could be represented by $adjacency\_list = \{0 : [1, [1, 4, 6, 7]], 1 : [0, [1, 4, 6, 7]]\}$. The memory space allocated to the interactions between nodes would be at least twice as large compared to the first encoding.

Other representations are possible such as a sequence of sparse adjacency matrices $(A_t)_{t\in T} \in \mathcal{M}_{n\times n}$, where an isolated temporal node $(t, u) \in W$ could be encoded by $A_t(u, u) = -1$, an absent node $(t, v) \notin W$ by $A_t(v, v) = 0$ and a temporal link $(t, uw) \in E$ by $A_t(u, w) = 1$. In the worst case, the according space complexity is in $O(\Omega N^2)$ making this encoding prohibitive, Ω being the number of event times.

Clusters

Numerous algorithms in Straph will output stream graph properties regarding a set of temporal nodes, a *cluster*. A *cluster* in Straph will always be encoded by an array of triples, corresponding to a set of node segments, (b, e, u). For instance, nodes in the stream graph S can be partitioned into clusters $C \subseteq W$ in which the degree of every temporal node $(t, v) \in C$ is identical. In this case, this partition into equal degree elements will be represented by a python dictionary associating a degree value to the corresponding cluster.

```
d = {1:[(1,4,'A'),(1,4,'B'),(6,7,'A'),(6,7,'B')],
     0:[(0,1,'A'),(4,5,'A'),(4,6,'B'),(7,10,'A'),(7,10,'B')]}
# This partition of W can be obtained using:
d = S.degrees_partition()
```

Connected Component Object

A *connected component* in Straph is a flexible structure covering several theoretical concepts such as strongly connected components (addressed in more details in

chapter 3).

A connected component is composed of a couple *times* corresponding to its time window, by a python set of nodes or segmented nodes (named *nodes*). An optional array of segmented links involving nodes in the set *nodes* can be added as well as an identifier, *id*, to deal with a decomposition of a stream graph, or a substream, into a set of connected components.

```
from straph import connected_component as comp
# A connected component corresponding to temporal nodes with a
   degree equal to 1
C = comp.connected_component(id=0,
                 times = [1,7],
                 nodes = {(1,4,'A'),(6,7,'A'),(1,4,'B'),(6,7,'B')}
                 )
```

DAG Object

Straph provides another useful data structure, a Directed Acyclic Graph (DAG) where nodes are *connected component* objects. A *DAG* object consists of an identifier *id*, a couple *times*, an array of *connected components* objects *c_nodes*, an array of links - a couple of *connected components* identifiers - *c_links* as well as a python dictionary linking an *id* to the corresponding *connected component* object. This structure and its applications are presented and detailed in chapters 4 and 5.

2.2.2 Streaming Formats

In order to manipulate huge stream graphs we provide a streaming format called **temporally ordered events**. Events are processed in a streaming fashion. This means that we read the data describing a stream graph in a time-ordered manner. We do not store it into central memory, and we output results as soon as they are available. Therefore, we do not store output in central memory either.

More precisely, we consider in input a time-ordered sequence of node or link arrivals or departures. For events occurring at a same time instant, we assume that arrivals are before departures, that node arrivals are before link arrivals, and that link departures are before node departures. In this way, if we read the input until a given event time t and then add artificial link and node departures at time t, the data we have processed until then is a consistent set of node and link segments.

In addition, we maintain the set of present nodes and links at the current time instant t, *i.e.* the graph G_t; in addition, we store their latest arrival time seen so far. This has a $\Theta(N + M)$ time and $\Theta(n + m)$ space cost for the whole processing of input data. Therefore, these worst-case complexities are lower bounds for our streaming algorithms.

In practice, this format consists in a temporally ordered sequence of events, represented by a tuple. In order to deal with simultaneous events, we associate a *code* to

each type of event. A node arrival is coded by 2, its departure by -2, a link arrival by 1 and its departure by -1. For instance, S, of Figure 1-1 will be encoded as follows:

```
# temporally ordered events
(2,0,'A')
(2,1,'B')
(1,1,('A','B'))
(-1,4,('A','B'))
(-2,5,'A')
(2,6,'A')
(1,6,('A','B'))
(-1,7,('A','B'))
(-2,10,'A')
(-2,10,'B')
```

2.2.3 File Formats

A wide range of real-world datasets are logged in *.csv* or *.tsv* files.

```
# stream_graph.tsv
1 A B
6 A B
```

Each line of these files is under the form $ts\ u\ v$ corresponding to an interaction between nodes u and v recorded at the timestamp ts. They may also record the duration of the interactions, under the form $ts\ l\ u\ v$, meaning that the interaction between u and v has started at time ts and finished at time $ts + l$. We provide flexible readers to parse these formats. It is possible to modify or input a new duration for the recorded interactions and to aggregate overlapping ones (see section 1.3).

Nonetheless, this format does not allow the storage of nodes' arrival and departure times. We propose another format *.sg* where a stream graph is written in two separate files: one for the presence times of nodes (*nodes.sg*) and another for the links and their interactions times (*links.sg*).

```
# nodes.sg
A 0 5 6 10
B 1 10

# links.sg
A B 1 4 6 7
```

2.3 Functionalities

Figure 2-1 presents an overview of a typical usage of Straph. Most available real-world datasets are stored in one of the formats supported by Straph, *csv*, *tsv*, *json* or *pcap*. Once a dataset has been parsed, we can apply Straph algorithms, extract

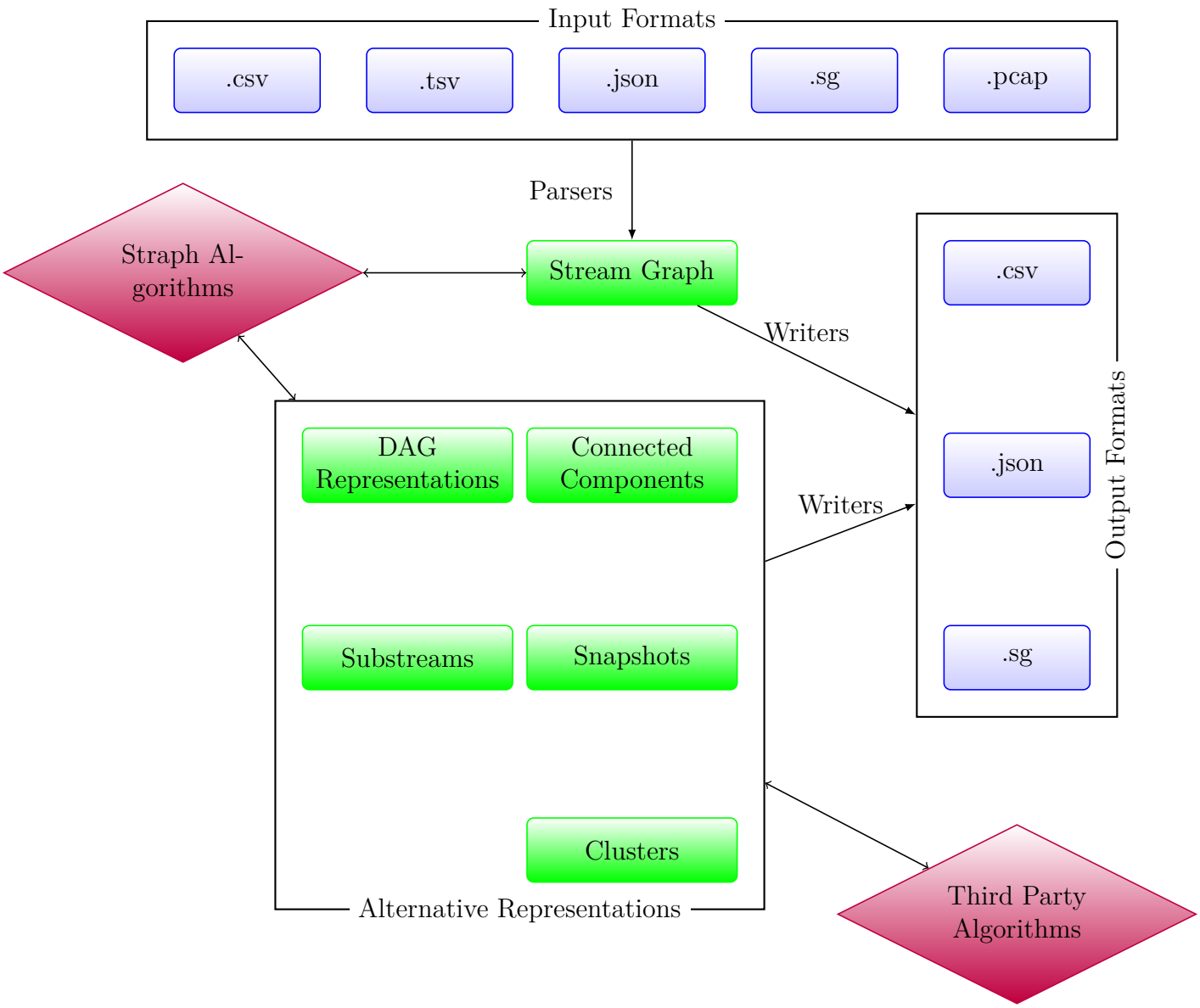

Figure 2-1: Diagram summing up Straph's functionalities

results under different kinds of representations - and apply third party algorithms, such as NetworX's ones, if it is a snapshot - these results can easily be outputted to common file formats such as *csv* or *json*.

Straph's API is similar to the one of NetworkX. A stream graph is easily editable.

```
S = stream_graph()
S.add_node('D',[2,3,4,5]) # We add 'C' from time 2 to 3 and 4 to 5
S.add_edge(('C','D'),[0,10]) # We add ('C','D') from time 0 to 10
```

Extracting a snapshot or an aggregate graph and analyse it with NetworkX is simple.

```
# Return the diameter of the snapshot at instant 2.5.
import networkx
S = stream_graph()
G = S.instant_graph(2.5, to_networkx=True)
networkx.diameter(G)
```

Straph provides a wide range of algorithms to compute basic and more complex features: degrees, k-cores, k-cliques, temporal paths, weakly connected components, strongly connected components, clustering coefficient, ... Several algorithms and their practical applications using Straph will be addressed in other chapters (3 and 5).

2.3.1 Installation and Dependencies

Straph is easy to install as long as a python environment is installed. We recommend using the Anaconda distribution. Straph can be installed using *pip* either from the python wheel or from Github (https://github.com/StraphX/Straph).

```
pip install straph
```

The main dependencies of Straph are Matplotlib, for the visualisations functionalities and Networkx for the support of (static) graph algorithms.

2.3.2 Visualisation

Straph provides several visualisations of stream graphs. Figure 2-2, created by the snippet of code below, allows a user to have a global view of a given stream graph.

```
S.plot()
S.plot_aggregated_graph()
```

The right part shows aggregated topology of the stream graph and the left one the dynamics of nodes and links.

As a stream graph is, by essence, an evolving structure, Straph also provides a dynamic visualisation consisting of two views side by side. One representing the whole stream graph, as in the first visualisation, accompanied by an animated cursor. The

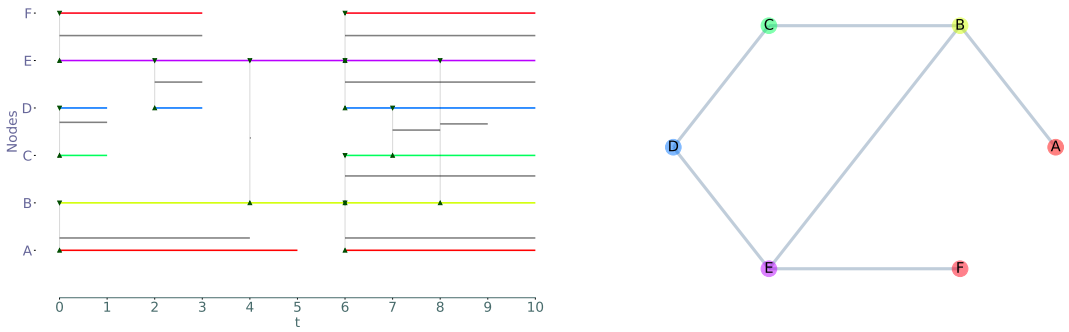

Figure 2-2: Straph drawings of a stream graph (left) and its aggregated graph (right)

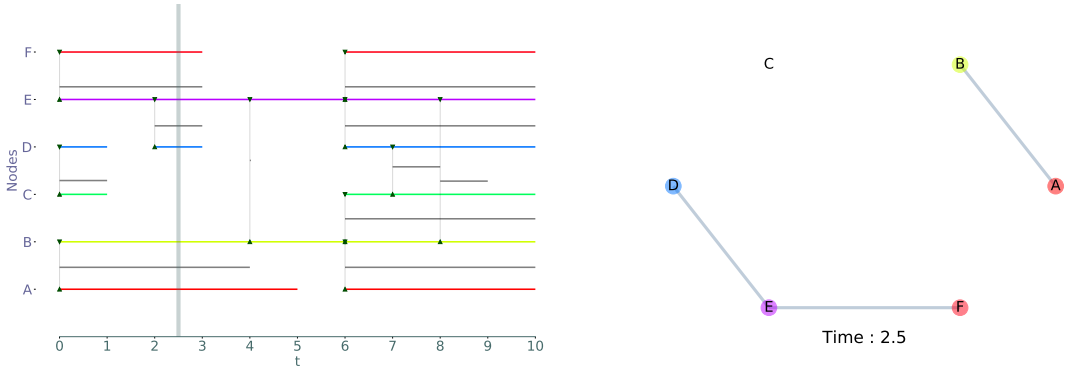

Figure 2-3: Illustration of an animated drawing of a stream graph. The left view features a moving cursor over the time axis on the stream graph's global drawing, here at $t = 2.5$. On the right view, an induced graph is drawn corresponding to the cursor's position.

second view represents the induced static graph at the time instant corresponding to the position of the cursor. An illustration of this visualisation is presented in Figure 2-3.

```
S.plot(animated=True)
S.plot_induced_graphs()
```

Numerous properties in Stream Graph properties assign a value to a node segment, its degree for instance, or to a group of node segments (a cluster) as the 3-core of a stream graph. Therefore we also provide visualisations for this purpose, as illustrated on Figure 2-4. It was obtained using the following snippet of code.

```
d = S.degrees_partition()
S.plot_dict_clusters(dict_clusters = d)

scc = S.strongly_connected_components(format="cluster")
S.plot(clusters = scc)
```

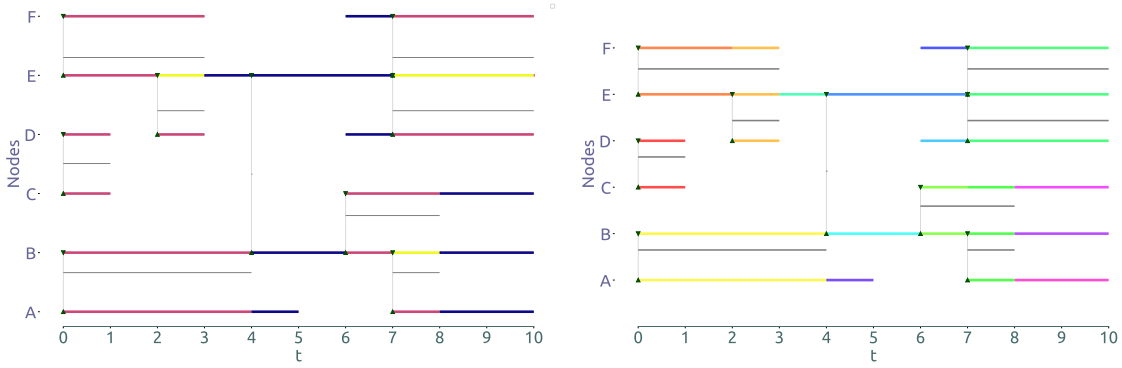

Figure 2-4: Illustrations of clustering visualisations in Straph. The instant degree value of each node (left) and clusters corresponding to the distinct strongly connected components of the stream graph (right).

As visualisations of stream graphs possessing many links tend to be messy, due to many crossing elements, we can disable links' display. Figure 2-5 shows the degree of temporal nodes in the *High School* dataset.

2.3.3 Straph Generators

Simulating random temporal graphs is a difficult task, we refer to [36] for a survey of existing methods. But no algorithm was provided to generate random stream graphs. We propose a simple way to adapt Erdős-Rényi and Barabási-Albert models for stream graphs.

The dynamic nature of stream graphs calls for a new procedure which consists in simulating random occurrences and durations of node and link segments. Occurrences of a node or a link, the number of corresponding node or link segments, follows a Poisson law. Arrivals and departures of node and link segments are randomly drawn following a uniform law with parameter (a, b), the time window of the stream graph. The duration of a node or link segment follows a uniform law. The probability determining the existence of a given link depends on the chosen model.

Our generators are quite complex, they need several parameters to generate a stream graph $S = (T, V, W, E)$:

- T: The time window of S.
- *nb_nodes*: The number of nodes composing V.
- *occurrence_param_node*: The parameter of the Poisson law determining the number of occurrences of a node, its number of segmented nodes.
- *presence_param_node*: The parameter of the Uniform law determining the duration of each segmented node.
- *occurrence_param_node*: The parameter of the Poisson law determining the

Figure 2-5: Straph drawing of the temporal nodes degree (brighter the color higher the degree) in a subset of the High School dataset (substream of the first fifty nodes on the second day of recording).

number of occurrences of a link, its number of segmented links.

- *presence_param_node*: The parameter of the Uniform law determining the duration of each segmented link.

In the following models, we aggregate overlapping nodes or links segments, we truncate any node or link segment exceeding the time window T and truncate link segments involving absent nodes.

Erdős-Rényi

In the Erdős-Rényi model, denoted by $G(n, p)$, a graph is constructed by connecting n nodes randomly with probability p. The main property of this model is that the distribution of a node's degree follows a binomial law.

This model leads to another parameter, *p_link*, which is the probability of a random link between two nodes in a stream graph. We verify the existence of a common presence interval between extremities of a randomly drawn link. If it exists we simulate its occurrences and presence times as defined previously otherwise we ignore it.

The following snippet of code shows how easy it is to simulate a random stream graph using Straph. Figure 2-6 shows an Erdős-Rényi like randomly generated stream graph.

```
from straph.generators import erdos_renyi

T = [0, 100]
nb_node = 21
occurrence_law_node = 'poisson'
presence_law_node = 'uniform'
occurrence_param_node = 3
presence_param_node = 25
occurrence_law_link = 'poisson'
presence_law_link = 'uniform'
occurrence_param_link = 5
presence_param_link = 15
p_link = np.sqrt(nb_nodes)/nb_nodes
S = erdos_renyi(T,
                nb_nodes,
                occurrence_law_node,
                occurrence_param_node,
                presence_law_node,
                presence_param_node,
                occurrence_law_link,
                occurrence_param_link,
                presence_law_link,
                presence_param_link,
                p_link)
```

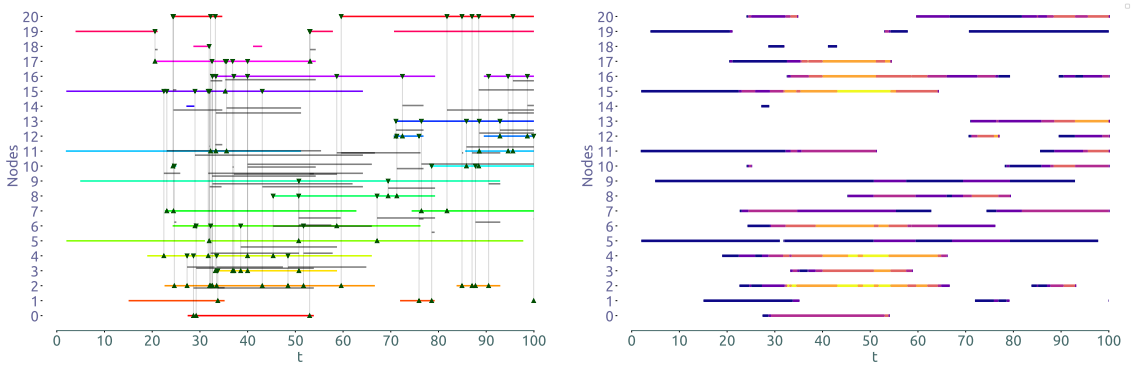

Figure 2-6: An Erdo-Rényi generated stream graph (left) along with the visualisation of the degree of its temporal nodes (brighter the color higher the degree)

The definition of a node's degree has been proposed in [59] (we refer to chapter 1 for more details):

$$d(v) = \frac{|N(v)|}{|T|} = \sum_{u \in V} \frac{|T_{uv}|}{|T|}$$

Figure 2-7 shows the nodes degree distribution in the Erdős-Rényi model. This distribution follows a Beta distribution, which is the continuous counterpart of the Binomial law, being consistent with graph theory results presented in [30].

Barabási-Albert

In the Barabási-Albert model, also called preferential attachment model, a graph begins with m_0 connected nodes. Then $n - m_0$ new nodes are added to the graph one at a time. Each new node will be connected to m - a parameter of the model - existing nodes with a probability proportional to their degree. The probability p_i that the new node is connected to the existing node i is: $p_i = \frac{d_i}{\sum_{j \in V} d_j}$, where d_i denotes the degree of the node i and V is the set of existing nodes. The resulting degree distribution is scale free: it follows a power law.

This model leads to two additional parameters m_0 and m. In this model we connect a new node to m existing nodes, therefore if a randomly drawn link cannot exist, due to the absence of a common presence interval between its extremities, we proceed to draw another link. Then, occurrences and presence times of links are randomly drawn as previously.

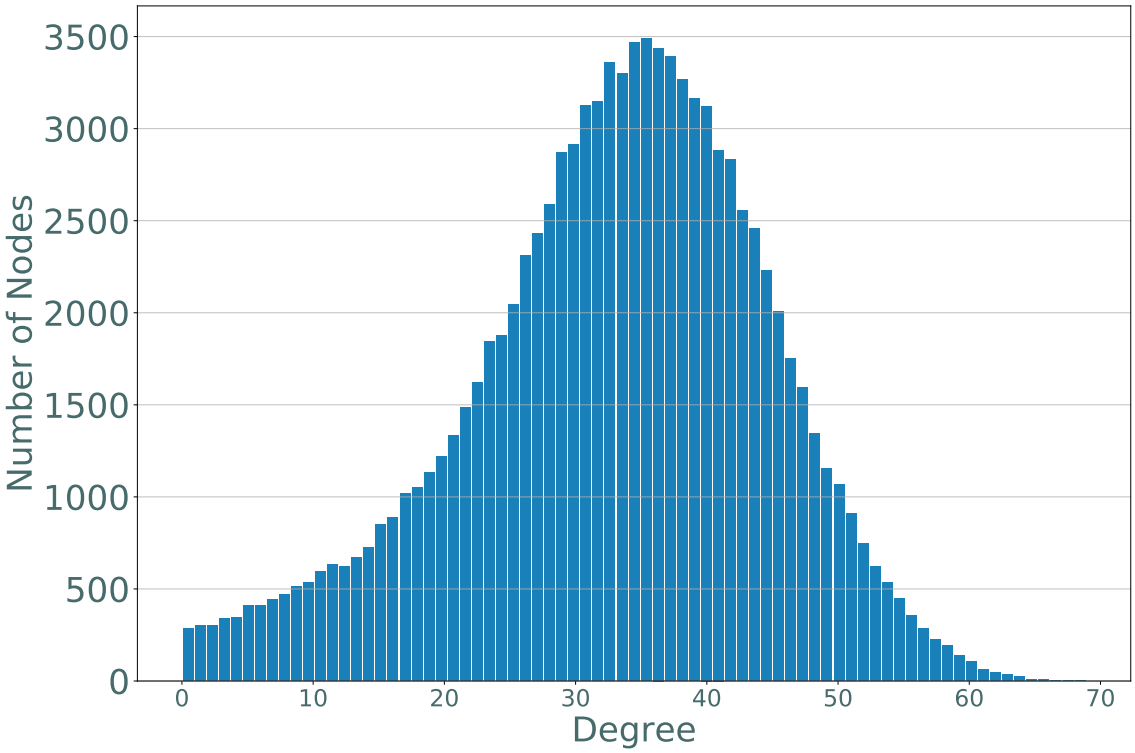

Figure 2-7: Degree distribution of a randomly generated Erdős-Rényi stream graph with parameters *T = (0, 1000), nb_nodes = 100000, occurrence_param_node = 4, presence_param_node = 200, occurrence_param_link = 3, presence_param_link = 150,* $p = \sqrt{nb\_nodes}/nb\_nodes$.

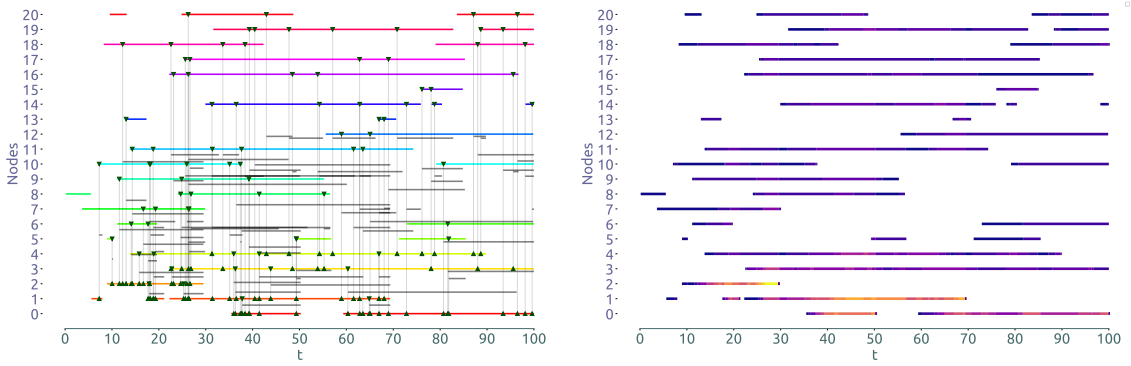

Figure 2-8: A Barabási-Albert generated stream graph (left) along with the visualisation of the degree of its temporal nodes (brighter the color higher the degree)

```
from straph.generators import barabasi_albert

T = [0, 100]
nb_node = 21
occurrence_law_node = 'poisson'
presence_law_node = 'uniform'
occurrence_param_node = 3
presence_param_node = 25
occurrence_law_link = 'poisson'
presence_law_link = 'uniform'
occurrence_param_link = 5
presence_param_link = 15
m0 = 3
m = 3
S = generators.barabasi_albert(T,
                  nb_nodes,
                  occurrence_law_node,
                  occurrence_param_node,
                  presence_law_node,
                  presence_param_node,
                  occurrence_law_link,
                  occurrence_param_link,
                  presence_law_link,
                  presence_param_link,
                  m0,
                  m)
```

Figure 2-9 shows the degree distribution in the Barabási-Albert model. This distribution follows a power law being consistent with graph theory results presented in [7].

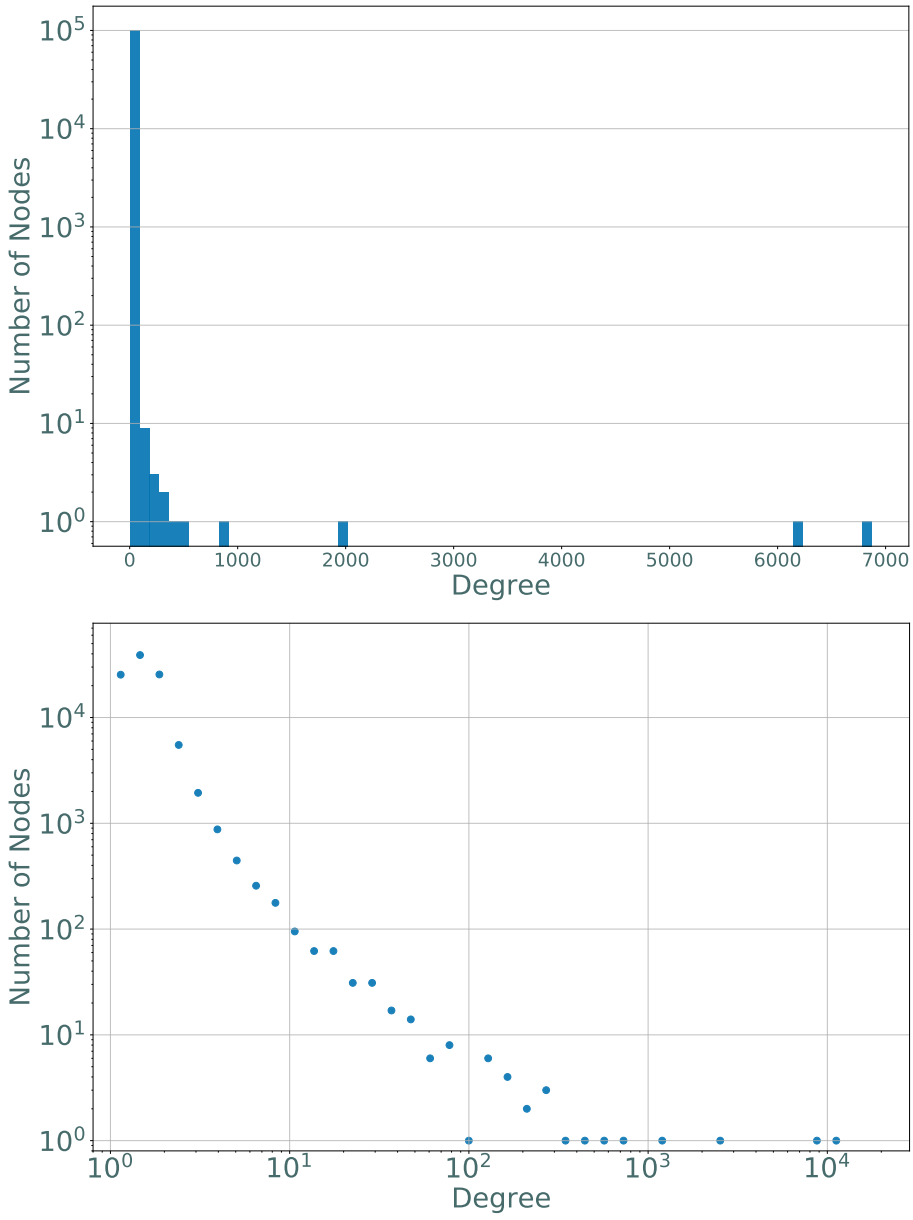

Figure 2-9: Degree distributions (top: histogram, bottom: log-log scatter plot) of a randomly generated Barabási-Albert stream graphs with parameters *T = (0, 1000), nb_nodes = 100000, occurrence_param_node = 3, presence_param_node = 500, occurrence_param_link = 3, presence_param_link = 250, m0 = 2, m= 2*

2.4 Real-World Use Case: High School Friends

Now, we demonstrate how useful Straph can be for the exploration of the **High School 2012 (HS 2012)** [33] dataset. It is the temporal network of interactions between students of 5 classes during 7 days in a high school in Marseille, France in 2012. An interaction consists in a physical proximity between two students, captured by a sensor. For more details on the processing of the data into a stream graph we refer to chapter 4.

Let us focus on the largest group of friends in this high school. We define this group as the largest group where each individual interacts with every other. We point out that we do not prioritize a period of time over another, break time and class time are considered equally. In other words, it is about finding the largest *clique* with the longest duration.

The definition of *clique*, for stream graphs, has been introduced in [59] and we recalled it in chapter 1.

```
# 'S' is a stream graph object containing the 'High School 2012'
    dataset
cliques = S.all_cliques()
# 'cliques' is a dictionary : 'clique size' -> 'clusters'
max(cliques)
-> 6
cliques[6]
-> [[(1386148020.0, 1386148040.0, 173),
  (1386148020.0, 1386148040.0, 125),
  (1386148020.0, 1386148040.0, 99),
  (1386148020.0, 1386148040.0, 169),
  (1386148020.0, 1386148040.0, 316),
  (1386148020.0, 1386148040.0, 176)]]
```

In this stream graph, the maximum clique size is 6 and there is only one 6-clique, with a 20 seconds duration. The longest 5-clique lasts for 140 seconds. These durations are too short relatively to the time window of the dataset, therefore we consider these cliques as non significant and focus on 4-cliques.

```
counter_clique = {}
for c in cliques[4]:
    t0,t1,_ = c[0]
    members = tuple(sorted([n for _,_,n in c]))
    if members in counter_clique:
        counter_clique[members] += t1-t0
    else:
        counter_clique[members] = t1-t0
suspects,duration = max(counter_clique.items(), key = lambda x:x[1])
[S.node_to_label[s] for s in suspects]
-> ['275', '312', '612', '886']
```

We found that a 4-clique occurs 59 times in S, for a total duration of 3300 seconds.

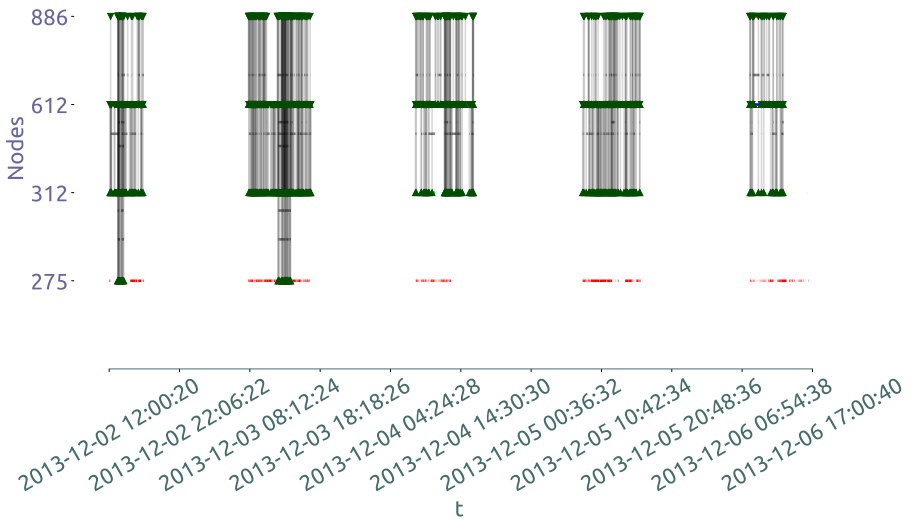

Figure 2-10: Substream induced by nodes *'275'*, *'312'*, *'612'*, *'886'* in the *High School* dataset.

It includes the following nodes: *'275'*, *'312'*, *'612'*, *'886'*. In Figure 2-10 we draw the substream induced by this clique. We observe that the node *'275'* does not have the same number of interactions as the other members of the clique. We can explain this difference by the fact that *'275'* is not in the same class as the others (confirmed by the metadata of the dataset).

The assumption that a group of friends is necessary a clique may be a bit too strong, given the conditions of the interactions' recording. Nevertheless, we can suppose that this group of friends gather frequently and when it happens they are connected with each other: they are in the same strongly connected component. Let us find the groups of nodes of at least 4 members which are strongly connected for the longest period of time.

```
scc = S.strongly_connected_components(format="cluster")
counter_scc = {}
for c in scc:
    t0,t1,_ = c[0]
    members = tuple(sorted([n for _,_,n in c]))
    if len(members) >= 4:
        if members in counter_scc:
            counter_scc[members] += t1-t0
        else:
            counter_scc[members] = t1-t0
suspects,duration = max(counter_scc.items(), key = lambda x:x[1])
[S.node_to_label[s] for s in suspects]
-> ['3', '884', '339', '147']
```

We found a group of 4 students, *'884'*, *'3'*,*'339'* and *'147'* that are connected with

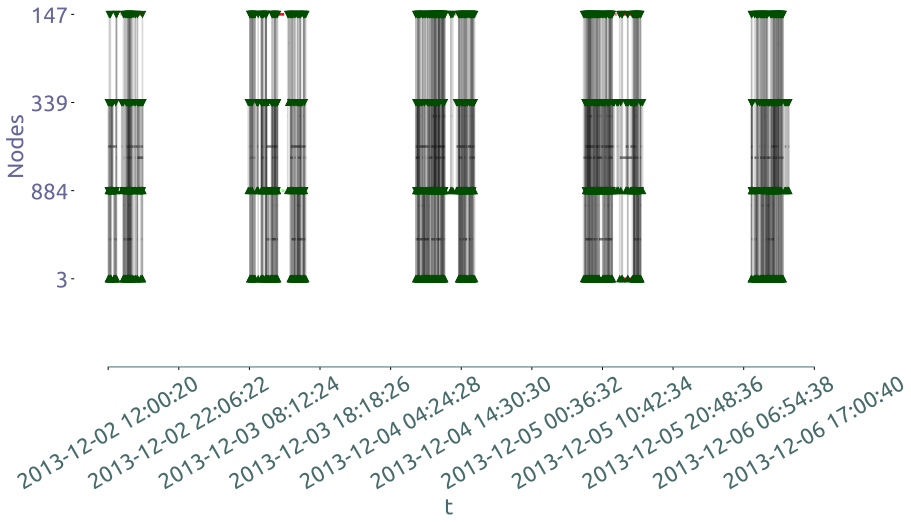

Figure 2-11: Substream induced by nodes *'884'*, *'3'*, *'339'* and *'147'* in the *High School* dataset.

each other for a total period of 9620s. In Figure 2-11, we observe that these students interact every day and that they are equally involved in the total number of interactions.

We have shown that with the help of Straph it is possible to bring an answer to this complex question with only a dozen lines of code. Other approaches based on Straph functionalities, such as k-cores, could have been used to address this problem. Another solution consists in finding the longest and densest substream. However this problem has a huge complexity and no known algorithm exists to solve it.

2.5 Discussion: development choices and future features

In this chapter, we have presented the numerous functionalities of Straph and its usefulness in practice. However many functionalities must be extensively tested and documented. Straph needs further development to be used in an industrial context.

In the future we aim to expand Straph in order to handle different kinds of stream graphs: with weighted links, where it takes time to cross a link, with a bipartite structure. The addition of other algorithms such as community detection, centrality measures (closeness, betweenness), links and nodes prediction or classifications, would render Straph sufficient to tackle many real word challenges.

3 Connectivity

Contributions

- Algorithms to compute connectivity notions in stream graphs
- Connectivity analysis of large scale real-world stream graphs

Connected components are among the most basic, useful and important concepts of graph theory. It is common usage to decompose a graph into its connected components. If a graph is not connected, it can be divided into distinct connected components. Many properties, which involve computation of paths or communities, can be computed independently on each connected component, thus enabling parallel execution of numerous methods.

Connected components were recently generalized to stream graphs [59]. These generalized connected components have a crucial feature: like graph connected components and unlike other generalizations available in the literature, they **partition** the set of temporal nodes. This means that each node at each time instant is in one and only one connected component. This makes these generalized connected components particularly appealing to capture important features of the vast variety of objects modeled by stream graphs.

However, actual computation of connected components in stream graphs has not been explored yet. Therefore, up to this date, they remain a formal object with no practical use. In addition, the algorithmic complexity of the problem is unknown, as well as the insight they may shed on real-world stream graphs of interest.

We introduce key notations and definitions and present an algorithm for weakly connected components in stream graphs (Section 3.1). Then we propose three algorithms for the more complex case of strongly connected components, together with their complexity (Section 3.2). We then apply these algorithms to several large-scale real-world datasets, in order to study their performances in practice, and to demonstrate their ability to describe such datasets (Section 3.3). After discussing related work in Section 3.4, we conclude this chapter and discuss potential improvements and

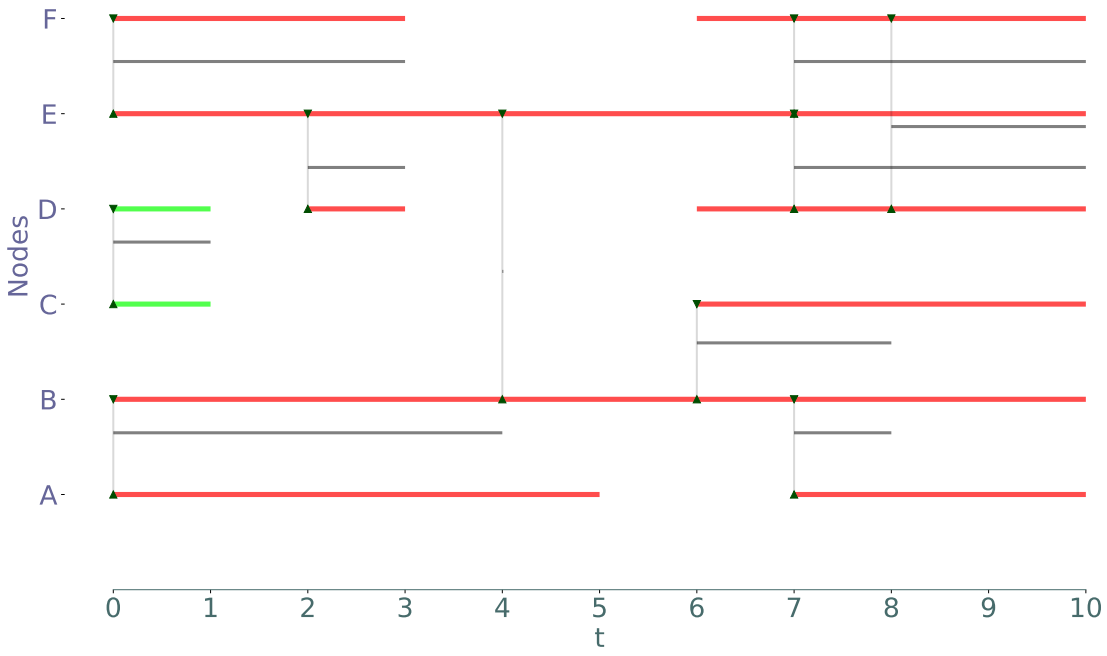

Figure 3-1: The two weakly connected components of the stream graph of Figure 1-1, each component having its own color.

extensions 3.5.

3.1 Weak Connectivity

In a Stream Graph, weakly connected components (WCC) represent elements of W connected together without any constraint on time.

Definition 3.1.1. *A **weakly connected component** of $S = (T, V, W, E)$ is a maximal subset C of W such that $\forall(\alpha, u)$, $(\omega, v) \in C$, there is a sequence (t_0, u_0, v_0), (t_1, u_1, v_1), …, (t_k, u_k, v_k) of elements of $T \times V \times V$ such that:*

$$u_0 = u \text{ and } [\min(\alpha, t_0), \max(\alpha, t_0)] \subseteq T_u$$

$$v_k = v \text{ and } [\min(t_k, \omega), \max(t_k, \omega)] \subseteq T_v$$

$$\forall i \in \{0, \dots, k-1\},\ v_i = u_{i+1},\ (t_i, u_i v_i) \in E \text{ and } [\min(t_i, t_{i+1}), \max(t_i, t_{i+1})] \subseteq T_{v_i}$$

Intuitively, the weakly connected components correspond to the disconnected parts of a drawing of a stream graph (see Figure 3-1). For instance, the stream graph of Figure 1-1 has two weakly connected components: $[0,1] \times \{C, D\}$ and $([0,5] \cup [7,10]) \times \{A\} \cup [0,10] \times \{B, E\} \cup [6,10] \times \{C, D, F\} \cup [2,3] \times \{D\} \cup [0,3] \times \{F\}$.

The weakly connected components form a partition of W. They can be computed easily from the graph $G_W = (\overline{W}, E_W)$ where two node segments $([b, e], u)$ and $([b', e'], v)$ in $\overline{W}$ are linked together if there is a link (t, uv) in E with $t \in [b, e] \cap [b', e']$. Indeed, the connected components of this graph are exactly the sets of node segments composing weakly connected components of S. These elements can be analysed separately to observe some properties, as we will show in the following.

Connected components of G_W may be obtained through any graph algorithm for this purpose. The simplest ones are search algorithms, like for instance depth-first search (DFS). They obtain the weakly connected components of S in $O(N + M)$ time and space, since G_W can be computed in this time and space, and the search cost is the same. We call this method *WCC DFS*.

An alternative is the *Union-Find* algorithm [34], that starts with isolated nodes, then processes the graph link by link and merges the connected components of its extremities computed so far. Here, we start with node segments, and we compute the weakly connected components by processing the stream graph link segment by link segment. Each link segment $([b, e], uv)$ connects two node segments $([b_v, e_v], v)$ and $([b_u, e_u], u)$, and we merge their connected components computed so far. One may consider link segments in the temporal order, thus obtaining a streaming algorithm. Like all other WCC streaming algorithms, though, we can output a WCC only after the end of all its node segments. Indeed, in the worst case, nodes may be present until the end of the stream, and then each WCC must stay in memory until the termination of the method. Its time complexity is $O(M \cdot \alpha(N))$, where $\alpha(.)$ is the inverse of Ackermann function, since it performs at most M unions of sets of size at most N, see [34] for details. Its space complexity is $O(N)$. We call this method *WCC UF*.

3.2 Strong Connectivity

Definition 3.2.1. *A **strongly connected component** of $S = (T, V, W, E)$ is a maximal subset $C = I \times X$ of W such that I is an interval of T and X is a connected component of G_t for all t in I. It is denoted by (I, X).*

This definition is consistent with the one used in graph theory: for any time instant, if we take the induced Graph G_t the SCC at t corresponds to the connected components of G_t. Using this definition one can observe that the set of all strongly connected components of S is a partition of W, see [59].

Inside a strongly connected component all nodes are reachable from any other at any time instant. This strong property means that many properties can be computed independently in each strongly connected component. This will enable the design of a parallel framework in Chapter 4.

In Figure 3-2, we represent the 17 strongly connected components of the stream graph of Figure 1-1. Notice that some component time intervals are closed, some are open

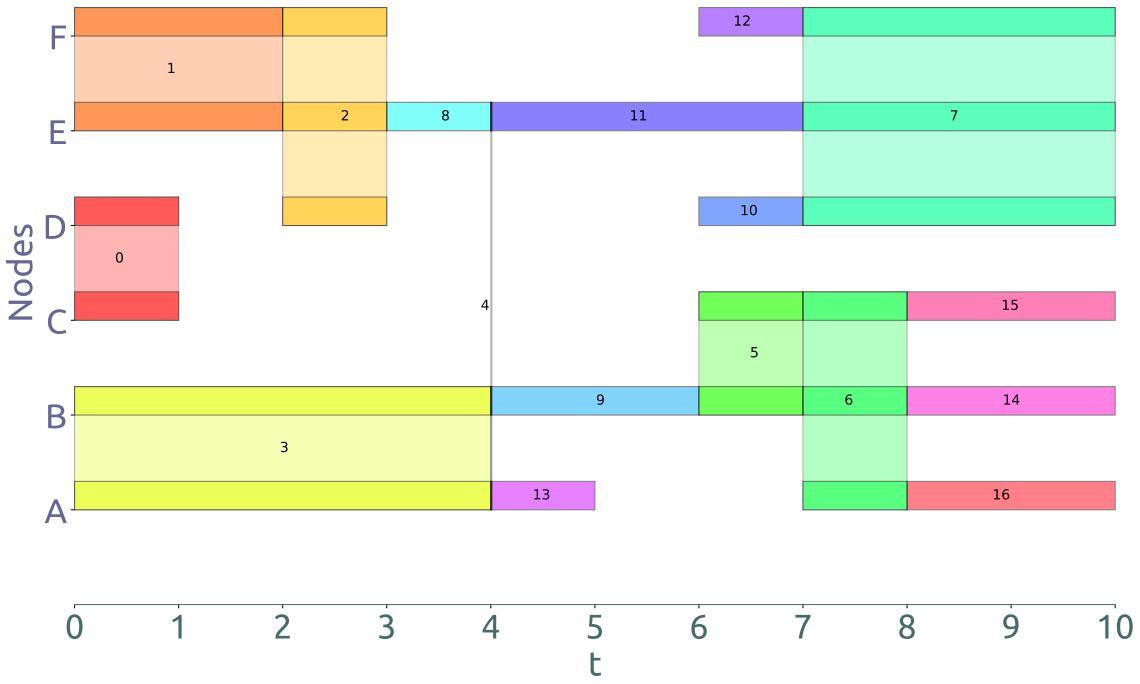

Figure 3-2: The 17 strongly connected components of the stream graph of Figure 1-1. Each component is numbered and has its own color.

and some are a combination of the two. For instance, $([0,1],\{C,D\})$ is a closed component, $(]4,6[,\{B\})$ is an open one, $([6,7[,\{B,C\})$ is a left-closed and right-open one, and $([4,4],\{A,B,C\})$ is a closed and instantaneous component.

Since the time intervals of components may be open or closed, we introduce the notation $\langle b,e\rangle$ to indicate an interval that can be either open or closed on its extremities. This interval contains $]b,e[$ and may or may not contain b and/or e. We will also use mixed notation: $\langle b,e]$ for instance designates an interval that may or may not contain b, but does contain e.

We present below several algorithms for computing the strongly connected components of a stream graph, using notations of Table 1.1 and the ones above.

Proposition 3.2.1. *The maximal number of strongly connected components in a stream graph is in* $O(N+M)$.

Proof. There can be one component per node segment, and each link segment may induce up to four components. Indeed, each beginning of link segment may correspond to the beginning of two components: one instantaneous at the link segment beginning and one that starts just after; and each link segment ending may correspond to the beginning of two connected components if the corresponding component becomes disconnected. □

Explicitly writing a component to the output is done in linear time with respect to its number of nodes; therefore we obtain the following.

Proposition 3.2.2. *Any algorithm explicitly listing the strongly connected components of a stream graph is in time* $\Omega(N + n \cdot M)$.

Proof. Consider for instance a stream graph $S = (T, V, W, E)$ with $W = T \times V$, and assume there is a node v such that all nodes of $V \setminus \{v\}$ belong to the same connected component of G_t for all t. Assume $\overline{E}$ is composed of $n-2$ link segments between nodes in $V \setminus \{v\}$ that last for all T, and many short link segments between v and another given node v' in V. In this case, the number of components is equal to the number of short link segments plus the number of intervals between these link segments, and each of these components will have either n or $n-1$ nodes. □

3.2.1 Direct Approach

One may compute strongly connected components directly from their definition, by processing event times in increasing order and by maintaining the set of strongly connected components that begin before or at current event time, and end after it. We represent each such component as a couple $(\langle b, C)$, meaning that it starts at b (included or not) and involves nodes in C.

More precisely, we start with a set $\mathscr{C}$ containing $([\alpha, C)$ for each connected component C of the graph G_α at the first event time α. Then, for each event time $t > \alpha$ in increasing order we consider the connected components of G_t^-. For each such component C, if there is no component $(\langle b, X)$ with $X = C$ in $\mathscr{C}$ then we add $(]t', C)$ to $\mathscr{C}$, where t' is the event time preceding t. For each element $(\langle b, X)$ of $\mathscr{C}$, if X is not a connected component of G_t^-, then we remove it from $\mathscr{C}$ and we output $(\langle b, t'], X)$. We then turn to the connected components of G_t: for each such component C, if there is no component $(\langle b, X)$ with $X = C$ in $\mathscr{C}$ then we add $([t, C)$ to $\mathscr{C}$; and for each element $(\langle b, X)$ of $\mathscr{C}$, if X is not a connected component of G_t, then we remove it from $\mathscr{C}$ and we output $(\langle b, t[, X)$. Finally, when the last event time $t = \omega$ is reached, we output $(\langle b, \omega], X)$ for each element $(\langle b, X)$ of $\mathscr{C}$.

Clearly, this algorithm outputs all strongly connected components of the considered stream graph. Computing the connected components of each graph is in $O(n + m)$ time and space. The considered set families (the graph connected components, as well as the elements of $\mathscr{C}$) form partitions of V. Therefore, their storage and all set comparisons processed for each event time have a cost in $O(n)$ time and space. There are $O(M + N)$ event times, therefore, the time complexity of this method is $O((N + M) \cdot (n + m))$, and it needs $O(n + m)$ space.

Without changing its time complexity, this algorithm may be improved by ignoring event times t such that all events occurring at t are link arrivals between nodes already in the same connected component. However, one still has to compute graph connected components at each event time with link departures. Therefore, this improvement is mostly appealing if many link departures occur at the same event times.

More generally, the approach above is efficient only if many events (node and/or links arrivals and/or departures) occur at each event time. Then, many connected components may change at each event time, and computing them from scratch makes sense. Instead, if only few events occur at most event times, managing each event itself and updating current connected components accordingly is appealing.

This leads to the following algorithm, which starts with an empty set $\mathscr{C}$, considers each event time t in increasing order, and performs the following operations.

1. For each node segment $([b, e], u)$ such that $b = t$ (node arrival), add $([b, \{u\})$ to $\mathscr{C}$.

2. For each link segment $([b, e], uv)$ such that $b = t$ (link arrival), let $C_u = (\langle b_u, X_u)$ and $C_v = (\langle b_v, X_v)$ be the elements of $\mathscr{C}$ such that $u \in X_u$ and $v \in X_v$; if $C_u \neq C_v$ then replace C_u and C_v by $([t, X_u \cup X_v)$ in $\mathscr{C}$. Then: if $\langle b_u \neq [t$ then output $(\langle b_u, t[, X_u)$; if $\langle b_v \neq [t$ then output $(\langle b_v, t[, X_v)$.

3. Let $G'_t = G_t$; then for each link segment $([b, e], uv)$ such that $e = t$ (link departure), let $C_v = C_u = (\langle b_u, X_u)$ be the element of $\mathscr{C}$ such that $u \in X_u$ and $v \in X_u$; remove the link uv from G'_t; if there is no path between u and v in G'_t then replace C_u by $C'_u = (]t, X'_u)$ and $C'_v = (]t, X'_v)$ in $\mathscr{C}$ where X'_u and X'_v are the connected components of u and v in G'_t, respectively; if $\langle b_u \neq]t$ then output $(\langle b_u, t], X_u)$.

4. For each node segment $([b, e], u)$ such that $e = t$ (node departure), let $C_u = (\langle b_u, X_u)$ be the element of $\mathscr{C}$ such that $u \in X_u$; remove C_u from $\mathscr{C}$; if $\langle b_u \neq]t$ then output $(\langle b_u, t], \{u\})$.

We call this algorithm *SCC Direct*. It clearly outputs the strongly connected components of the considered stream, like the previous algorithm.

It performs $2(M + N)$ of the steps above, corresponding to N node arrivals and departures and M link arrivals and departures. One easily deals with node arrivals and departures in constant time. If a link arrival induces a merge between two components, the smallest component may contain up to $n/2$ nodes and computing their union is in $O(n)$, as is outputting both components if needed. Thus the complexity for link arrival steps is in $O(M \cdot n)$. Each link departure calls for a computation of the connected components of a graph, and writing a component to the output is in $O(n)$. Thus the complexity for link departure steps is in $O(M \cdot (m + n))$. We obtain a total time complexity in $O(M \cdot (m + n) + N)$.

The space complexity is still in $O(n+m)$ as above, and we obtain the following result.

Proposition 3.2.3. *SCC Direct computes the strongly connected components of a stream graph in $O(M \cdot (n + m) + N)$ time and $O(m + n)$ space.*

3.2.2 Fully Dynamic Approach

The SCC Direct algorithm presented above is strongly related to one of the most classical algorithmic problems in dynamic graph theory, called fully dynamic connectivity [53, 110, 51, 8, 48, 52, 46], which aims at maintaining the connected components of an evolving graph. More precisely, dynamic connectivity algorithms consider a sequence of link additions and deletions, and maintain a data structure able to tell if two nodes are in the same connected components (*query* operation) and to merge or split connected components upon link addition or removal (*update* operation).

This data structure and the corresponding operations can be used in the above algorithm: we can use the data structure to store $\mathscr{C}$, the set of current connected components (we also need to store the beginning time of each component, which has negligible cost). Then, at each link arrival or departure, we can use the query operation to test whether the two nodes are in the same component or not, and the update operation to add or remove the current link to the data structure, while keeping an up-to-date set of connected components. When we observe a node appearance it is necessarily isolated, so we have to add the current time to its component. All the other steps (mainly, writing the output) are unchanged. We call this algorithm *SCC FD*.

Several methods efficiently solve the dynamic connectivity problem, the key challenge being to know if updates and queries may be performed in $O(\log(n))$ time, where n is the number of nodes in the graph. Current exact solutions perform updates in $O\left(\sqrt{\frac{n\cdot(\log\log(n))^2}{\log(n)}}\right)$ worst time [53], or in $\frac{\log^2(n)}{\log\log(n)}$ amortized worst time [110]. Probabilistic (exact or approximate) methods perform even better, but they remain above the $O(\log(n))$ time cost [48, 52, 46].

It is well acknowledged that these algorithmic time and space complexities hide big constants, and that the underlying algorithms and data structures are very intricate. As a consequence, implementing these algorithms is an important challenge in itself [8, 51], and the results above should be considered as theoretical bounds. In practice, the implemented algorithms typically have $O(\log(n)^3)$ amortized time and linear space complexities, still with large constants [8, 51].

In SCC FD, we perform $O(M)$ updates and queries, which leads to a $O(M\cdot\text{polylog}(n))$ overall time cost for these operations, with any of the polylog dynamic connectivity algorithms cited above. This cost is dominated by the cost of outputting the results, which is in $O(M\cdot n)$. An additional N factor is needed to deal with node arrivals and departures. The space cost of dynamic connectivity methods is in $O(m+n\cdot\log n)$, and we do not store significantly more information. We therefore obtain the following result.

Proposition 3.2.4. *SCC FD computes the strongly connected components of a stream graph in $O(M\cdot n+N)$ time and $O(m+n\cdot\log n)$ space.*

This algorithm is particularly appealing if large connected components are quite sta-

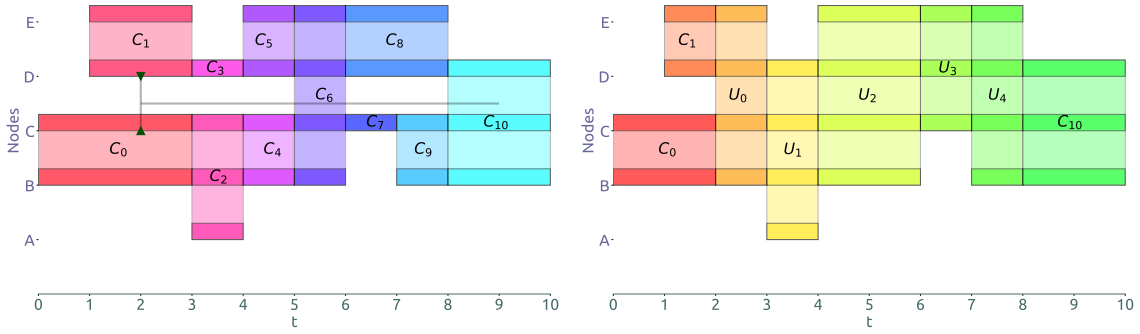

Figure 3-3: *SCC UF* algorithm illustration: The addition of a link ([2, 9],CD) between C and D causes component unions (U_0, U_1, U_2, U_3 and U_4). Components C_0 and C_1 are split at the begin of the link (instant 2) and component C_{10} remains unchanged.

ble, *i.e.* if most of the largest strongly connected components in the stream have a long duration. Indeed, in this case, fully dynamic update operations are much faster than updates used in SCC Direct, and the output is much smaller than the maximum $\Omega(N + M \cdot n)$ bound. The cost of SCC FD is then dominated by fully dynamic operations, and its time complexity is reduced to $O(M \cdot \text{polylog}(n))$.

3.2.3 Union-Find Approach

We propose here another approach, inspired by the classical union-find graph algorithm. It starts with each node segment as a strongly connected component in its own, and it handles each link segment one by one by merging and splitting previously computed connected components. See Figure 3-3 for an illustration.

We start with a set $\mathscr{C}$ that contains $([b, e], \{v\})$ for all node segment $([b, e], v)$ in $\overline{W}$ and set $I = [b, e]$; then for each link segment $([b, e], uv)$:

1. we consider the elements $C_u = (I_u = \langle b_u, e_u \rangle, X_u)$ and $C_v = (I_v = \langle b_v, e_v \rangle, X_v)$ of $\mathscr{C}$ such that $(b, u) \in X_u \times I_u$ and $(b, v) \in X_v \times I_v$; then
   - if $C_u = C_v = (I_{uv}, X_{uv})$ then set $I = [b, e] \setminus I_{uv}$; if $I = \emptyset$ then there is nothing to do because u and v already belong to the same component during the whole interval $[b, e]$; stop here
   - if $C_u \neq C_v$ and $b_u < b$ then replace C_u by $(I_u \setminus [b, e], X_u)$ and $(I_u \cap [b, e], X_u)$ in $\mathscr{C}$ (if $I_u \setminus [b, e]$ is composed of two maximum intervals I_1 and I_2 because $[b, e] \subsetneq I_u$ then add (I_1, X_u) and (I_2, X_u) to $\mathscr{C}$)
   - if $C_u \neq C_v$ then proceed likewise for C_v
2. we consider the elements $C_u = (I_u = \langle b_u, e_u \rangle, X_u)$ and $C_v = (I_v = \langle b_v, e_v \rangle, X_v)$ of $\mathscr{C}$ such that $(e, u) \in I_u \times X_u$ and $(e, v) \in I_v \times X_v$; then
   - if $C_u = C_v = (I_{uv}, X_{uv})$ then set $I = I \setminus I_{uv}$

- if $C_v \neq C_u$ and $e < e_u$ then replace C_u by $(I_u \cap [b, e], X_u)$ and $(I_u \setminus [b, e], X_u)$ in $\mathscr{C}$;
- if $C_u \neq C_v$ then proceed likewise for C_v

3. let L_u be the list of components $(I_u, X_u) \in \mathscr{C}$ such that $I_u \subset I$ and $u \in X_u$; let L_v be the list of components $(I_v, X_v) \in \mathscr{C}$ such that $I_v \subset I$ and $v \in X_v$; both lists are sorted by the natural order of their (open or closed) intervals; notice that the set of the intervals of the components of either list is a partition of I.

   While both lists are non-empty, take and remove their first elements C_u and C_v. By construction, these components both begin at the same time and their interval are both open or both closed on the left; i.e. we have $C_u = (I_u = \langle b_{uv}, e_u\rangle, X_u)$ and $C_v = (I_v = \langle b_{uv}, e_v\rangle, X_v)$, and either $b_{uv} \in I_u$ *and* $b_{uv} \in I_v$, or $b_{uv} \notin I_u$ *and* $b_{uv} \notin I_v$;

   - if $I_u \neq I_v$ then without loss of generality we can assume $I_u \subset I_v$; then change C_v to $C_v = (I_u, X_v)$ and add $(I_v \setminus I_u, X_v)$ at the beginning of L_v.

   Notice now that C_u and C_v cover exactly the same interval $I_{uv} = \langle b_{uv}, e_{uv}\rangle$; then

   - if $C_u \neq C_v$ then add their union $(I_u, X_u \cup X_v)$ to $\mathscr{C}$.

   Now we have that u and v belong to the same element $C_{uv} = (I_{uv}, X_{uv})$ of $\mathscr{C}$. At this point this element may have the same vertex set than the element just before; we then have to merge their time intervals:

   - let $C_u^p = (I_u^p, X_u^p)$ be the element preceding C_{uv} for u, i.e. the one such that $u \in X_u^p$ and I_u^p is just before I_{uv} in the natural ordering of all elements containing u. If $X_u^p = X_{uv}$ then we remove it and C_{uv} from $\mathscr{C}$ and we add $(I_u^p \cup I_{uv}, X_{uv})$ to $\mathscr{C}$.

   Finally the element just after the last one we dealt with may have the same vertex set; we also merge their time intervals:

4. let $C_u^f = (I_u^f, X_u^f)$ be the element of $\mathscr{C}$ following C_{uv} for u, i.e. the one such that $u \in X_u^f$ and I_u^f is the one just after I_{uv} in the natural ordering of all the elements of $\mathscr{C}$ containing u. If $X_u^f = X_{uv}$ then we remove it and C_{uv} from $\mathscr{C}$ and we add $(I_u^f \cup I_{uv}, X_{uv})$ to $\mathscr{C}$.

Figure 3-3 illustrates one step of the algorithm: the addition of link ([2, 9],CD). It causes $C_1 = ([1, 3], X_1)$ to be split into $([1, 2[, X_1)$ and $([2, 3[, X_1)$; $C_0 = ([0, 3], X_0)$ to be split into $(]0, 2[, X_0)$ and $([2, 3[, X_0)$; I is equal to $[2, 9[$; L_C contains components $([2, 3[, X_1)$, C_3, C_5, C_6 and C_8; and L_D contains components $([2, 3[, X_0)$, C_2, C_4, C_6, C_7 and C_9.

The space complexity of SCC UF is $O(N + n \cdot M)$ since, as explained above, there may be at most N components induced by node segments, and each link segment may correspond to the beginning of at most four components, each containing no more than n nodes.

The initialisation step takes $O(N)$ time. For each considered link segment $([b, e], uv)$, Steps 1 and 2 involve replacing one element of $\mathscr{C}$ with two elements, both with the same node set as the original. This may be done in $O(n)$. Step 4 involves a set comparison, which may also be done in $O(n)$. In Step 3, the number of elements of L_u and L_v may be in $O(M)$ (u and v each belong to a single node segment during the whole interval $[b, e]$). Finding the first elements of L_u and L_v may be done in $O(\log M)$ by logarithmic search. At each iteration of the loop over the first elements of L_u and L_v, one element is removed from each list, and at most one is added to one of those lists, so the number of element pairs considered in this loop is at most $|L_u| + |L_v|$. The union of the (disjoint) node sets of two elements of $\mathscr{C}$ may be done in linear time with their sizes, so in $O(n)$, and replacing an element with two copies of itself as well. Finally, the total number of operations performed in Step 3 is in $O(n \cdot M^2)$, leading to the following result.

Proposition 3.2.5. *Algorithm SCC UF computes the strongly connected components of a stream graph in $O(N + n \cdot M^2)$ time and $O(N + n \cdot M)$ space.*

3.3 Experiments and Applications

In this section, we first present experiments that show the practical usability of our algorithms in real-world cases. We also present an in-depth analysis of a typical large-scale real-world case, and apply our algorithms to approximate latency computations.

We publicly provide Python 3 implementations of our algorithms in the Straph library [79] (see Chapter 2). We used these implementations for all our experiments, and ran them on an Intel Core i5 3.4 GHz CPU and 32 GB RAM Linux machine. The following snippet of code illustrates a practical usage in Straph.

```
S = stream_graph()
# The algorithm used to compute WCC is "WCC DFS"
wcc = S.weakly_connected_components(format = "cluster")
# "SCC-Direct" is the default algorithm
scc = S.strongly_connected_components(format = "cluster",
                                      method = "Direct")
# SCC can also be computed using "SCC FD" or "SCC UF" algorithms
scc = S.strongly_connected_components(format = "cluster",
                                      method="FD")
scc = S.strongly_connected_components(format = "cluster",
                                      method="UF")
```

3.3.1 Algorithm performances

We conducted thorough experiments with the datasets described in Section 1.3 and our different algorithms. In all cases, **SCC Direct was significantly faster than others, and in many cases only SCC Direct was able to perform the computation in main memory. It was able to handle datasets of several dozens**

	$\lvert\mathscr{W}\rvert$	$\lvert\mathscr{C}\rvert$	*WCC UF*	*SCC Direct*	*SCC FD*	*SCC UF*
UC	11K	54K	0.35	**0.47**	6.83	25.14
HS 2012	13K	50K	0.45	**0.41**	5.11	6.34
Digg	26K	144K	1.0	**1.34**	19.4	78.89
Infectious	16.9K	106.3K	0.80	**1.18**	16.0	18.1
Twitter	113K	580K	3.94	**33.89**	9.98K	-
Linux	63K	698K	3.8	**11.6**	227.0	-
Facebook	373K	795.0K	10.36	**7.96**	55.3	109.23
Epinions	75.6K	75.6K	5.6	**3.3**	-	-
Amazon	4.1M	4.1M	117.3	**102.3**	691.1	-
Youtube	738.6K	1.2M	96.2	**150.8**	-	-
Movielens	282K	12.5M	93.3	**340.0**	-	-
Wiki	4.2M	23.5M	247.34	**247**	-	-
Mawilab	1.1M	30M	430	**90K**	-	-
Stackoverflow	5.2M	45.9M	410.7	**36.7K**	-	-

Table 3.1: Number of WCC ($\lvert\mathscr{W}\rvert$) - Number of SCC ($\lvert\mathscr{C}\rvert$) - Algorithms running time in seconds (K = 10^3, M=10^6)

of millions of link segments. Instead, both SCC UF and SCC FD have prohibitive memory space requirements, at least in their current versions. Notice that, even in the cases where SCC FD was able to handle the data, the constants hidden in its time complexity made it slower than SCC Direct. We therefore focus on SCC Direct here.

Table 3.1 shows the number of weakly and strongly connected components as well as the running times of connectivity algorithms. (The exact numbers of WCC and SCC in the stream graphs of the first twelve datasets are presented in the Table 4.1 of Chapter 4).

Figure 3-4 presents the time cost for each dataset, and show a strong relation between the number of link segments, the number of connected components, and computation time. Notice, however, that *Wiki* and *Mawilab* have similar numbers of link and node segments but *SCC Direct* is several orders of magnitude faster on *Wiki*. This difference comes from their quite different structure regarding connected components: *Mawilab* has more than $21M$ SCC involving at least $30K$ nodes, whereas *Wiki* has only $2K$ such SCC. As explained in Section 3.2, the computational cost of *SCC Direct* mainly depends on the number of nodes in each SCC, which is observed in this experiment.

Going further, Figure 3-5 displays the relations between the number of strongly connected components and other features of interest. It shows that the running time depends strongly on the number of strongly components, but other factors play a role, as explained above for *Wiki* and *MawiLab*.

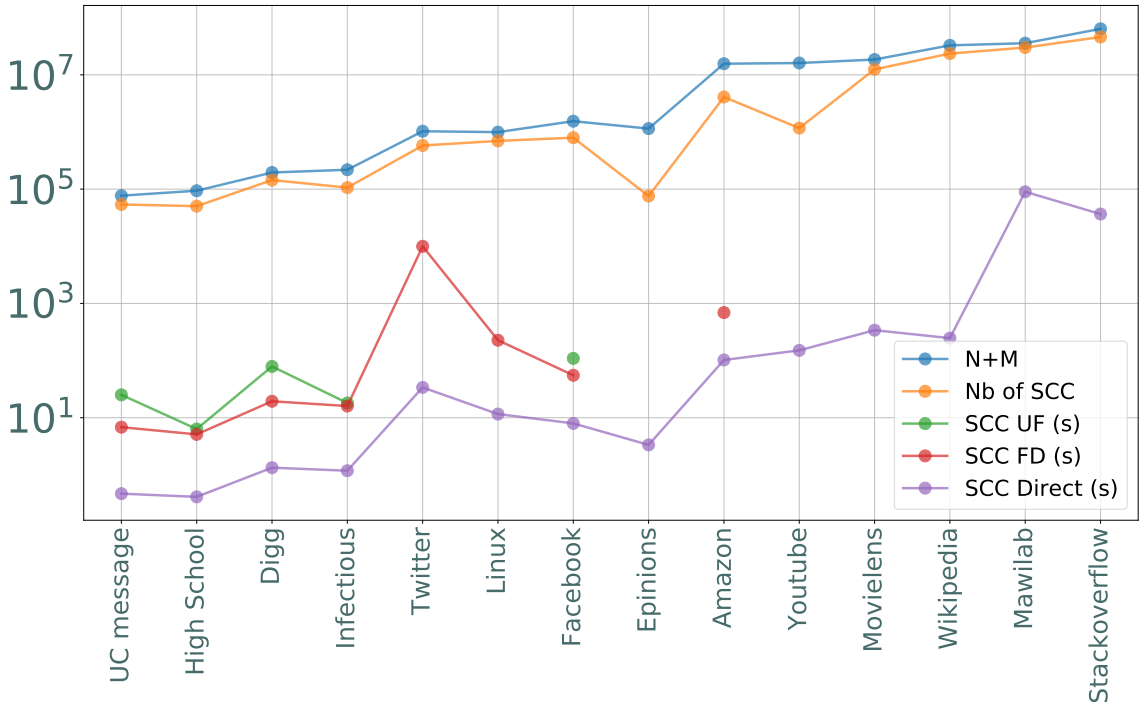

Figure 3-4: Time cost of *SCC Direct*, *SCC UF*, *SCC FD* in seconds, along with the number M of link segments and the number of strongly connected components, for each considered stream (horizontal axis, ordered with respect to M).

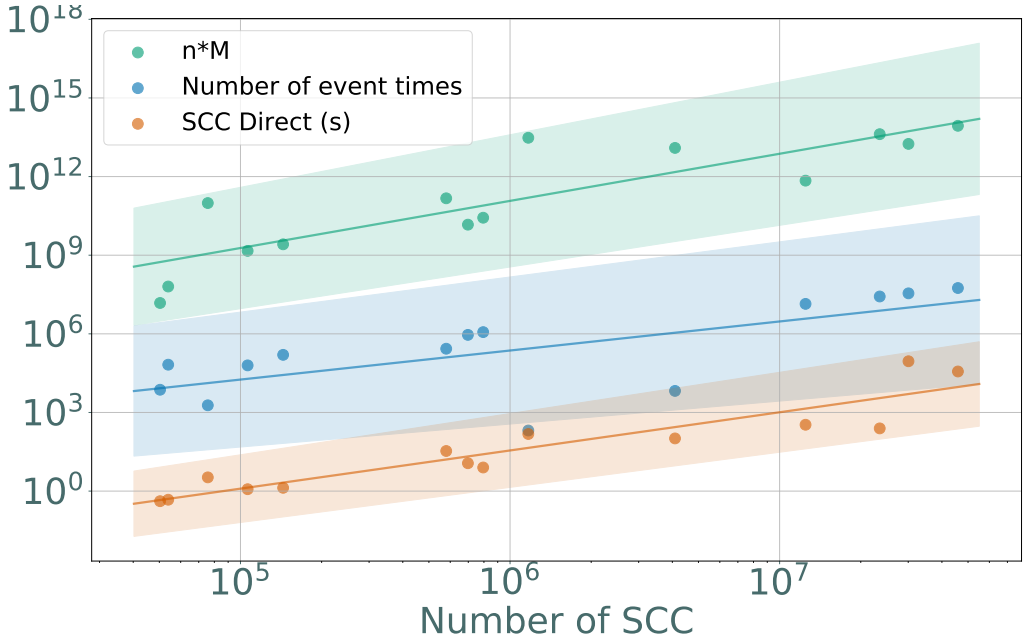

Figure 3-5: Relation between the number of strongly connected components (horizontal axis) and $n * M$, the number of event times, and the running time of *SCC Direct*. Each dataset leads to three vertically aligned points, the color of which indicating the considered variable.

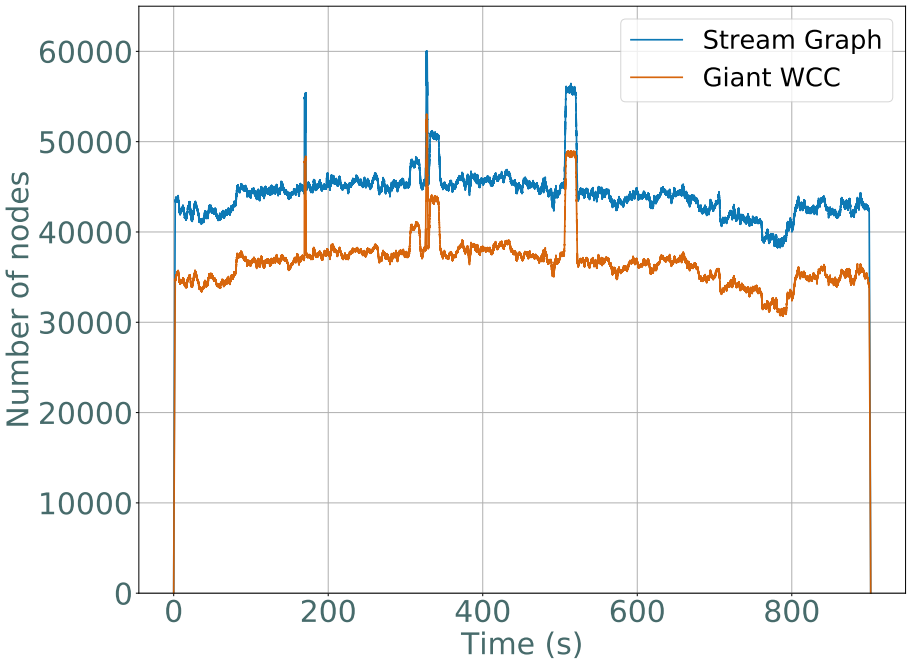

Figure 3-6: Number of nodes in the whole stream graph and in the giant weakly connected component over time in the *mawilab* dataset.

3.3.2 Connectedness analysis of IP traffic

We take the MawiLab IP traffic capture as a typical instance of large real-world dataset modeled by a stream graph, and we use it to illustrate the relevance of connected component analysis.

First notice that this stream graph contains a giant weakly connected component (see Figure 3-6), spanning 82.86% of W, involving 73.20% of all nodes in V and spanning the whole T duration. The second largest weakly connected component represents only 2.57% of W, and the $1,138,096$ other ones account for 14.57% of W all together. This extends to stream graphs the classical observation that real-world undirected complex networks generally have a giant connected component [71].

As we can observe in Figure 3-6 for nodes, if we observe a given dynamic over time in the stream graph - several surges in the number of nodes for instance - this dynamic can be solely connected to the dynamic of the giant weakly connected component. Therefore dynamics present in smaller WCC can be totally eclipsed in an overall observation, thus motivating the need to analyse each WCC independently.

While the number of nodes fluctuates greatly over time, the number of SCC existing at a given instant remains stable, as observed in Figure 3-7.

The situation of strongly connected components is quite different. Indeed, the stream has $30,062,184$ such components, with no giant one. Let us define the span $\sigma(C)$ of a connected component $C = (\langle b, e\rangle, X)$ as the fraction of W that it contains:

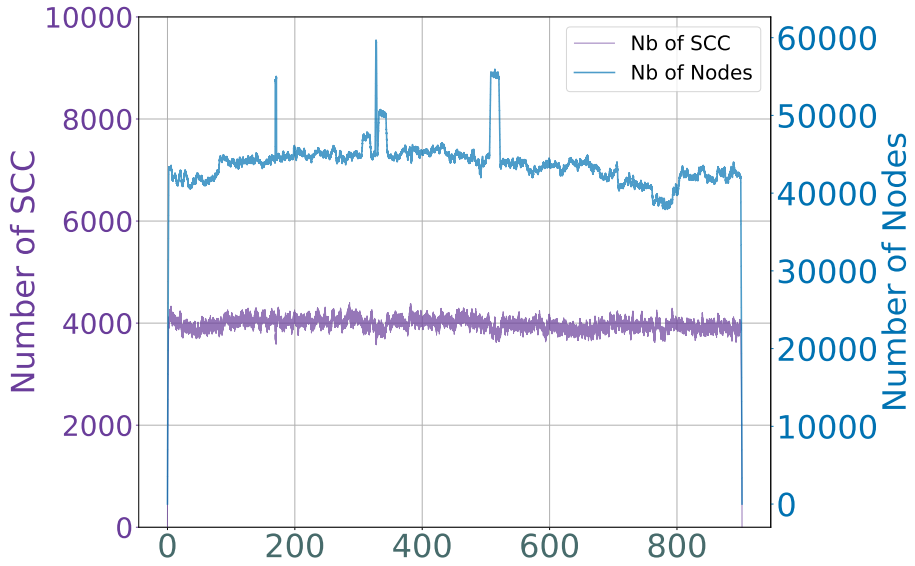

Figure 3-7: Number of nodes and number of strongly connected components over time in the *mawilab* dataset.

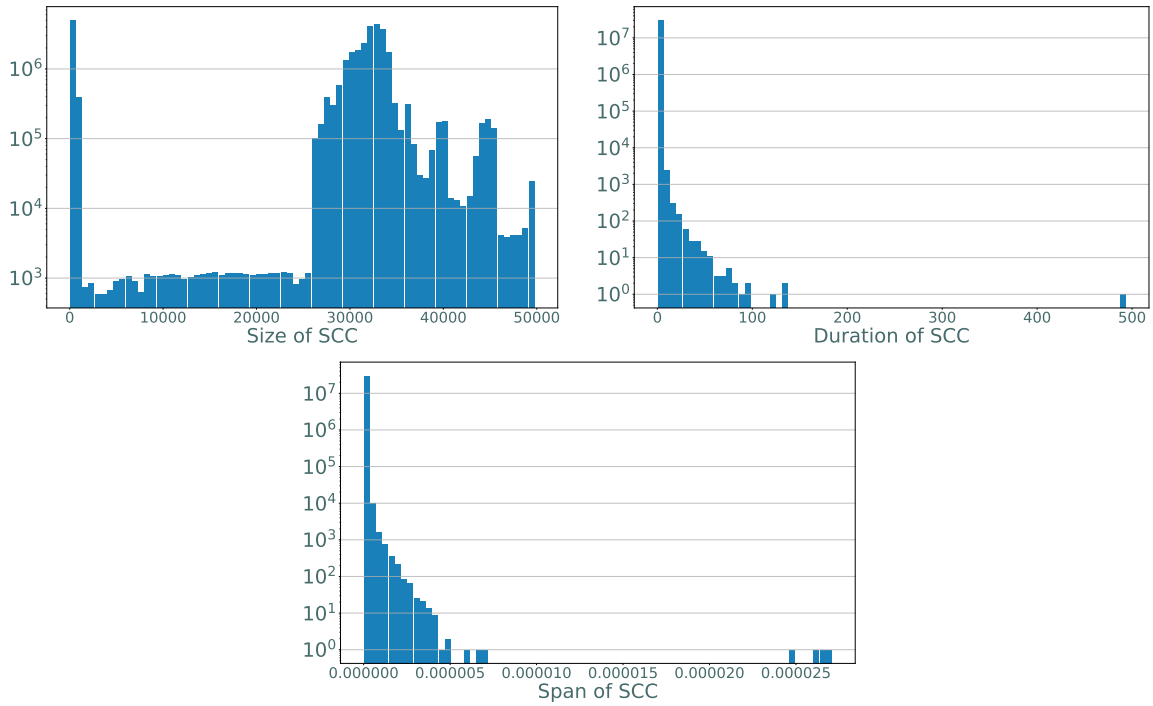

Figure 3-8: Distribution of the size (top left), duration (top right) and span (bottom) of strongly connected components in *Mawilab*.

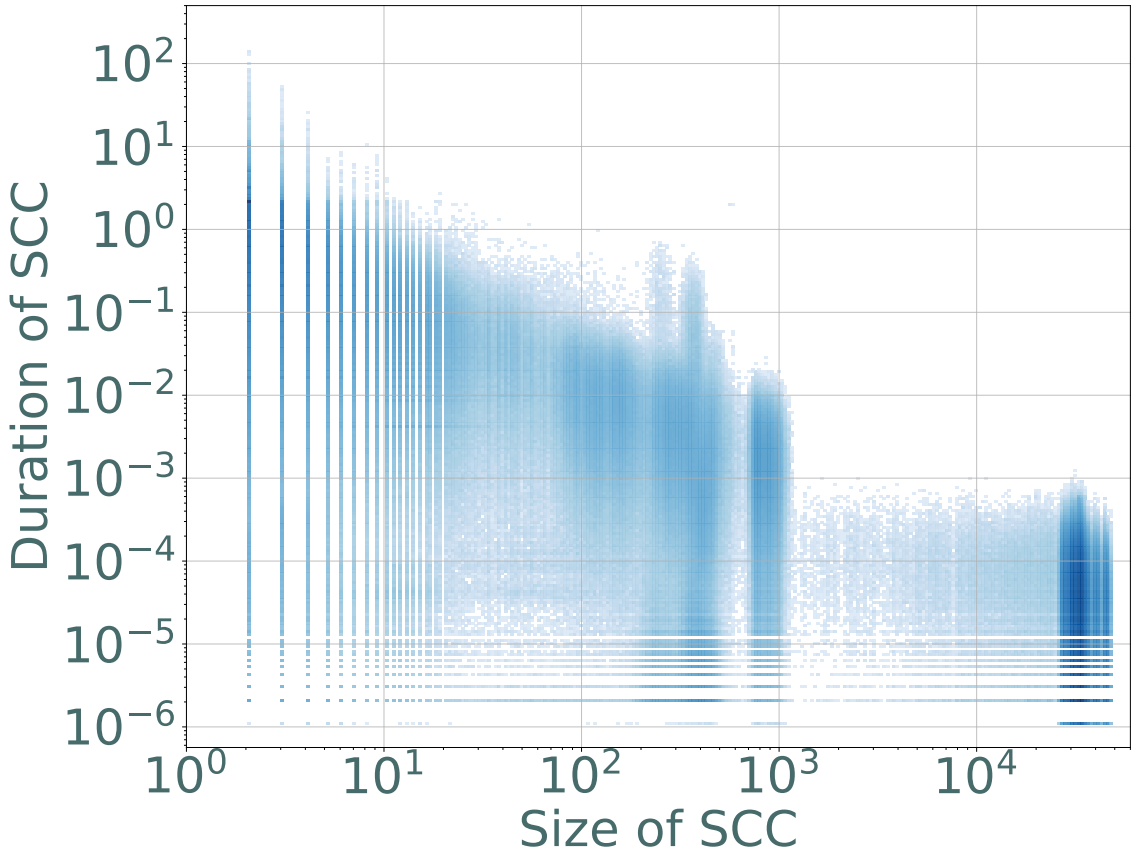

Figure 3-9: Duration of each strongly connected component as a function of its size in *Mawilab* dataset, in log-log scales. We added 10^{-6} to each duration in order to display instantaneous SCC on a log scale.

$\sigma(C) = \frac{(e-b)\cdot|X|}{|W|}$. We call the number of involved nodes $|X|$ its size, and the length of its time interval $e - b$ its duration. We display in Figure 3-8 the strongly connected component span distribution, as well as the size and duration distributions.

These distributions clearly show that the span of strongly connected components is very limited, the highest being 0.000027 of W. Many small components exist, but several have a span quite close to this maximal: 7 have a span larger than 0.0000049, for instance. Therefore, no component significantly stands out of the crowd. This is confirmed by component size and duration distributions: the largest component (in terms of number of nodes) involves $49,791$ nodes (only 5.3% of the whole), and lasts for 0.000082 seconds (only 0.0000091% of the whole).

We also display in Figure 3-9 the duration of components as a function of their size. The correlation coefficient between size and duration of SCC is -0.15, therefore they are not linearly dependant. However, Figure 3-9 clearly shows that there is no component with both a long duration and a large size; instead, large strongly connected components have a very short duration, and, conversely, long components

have a small size. For instance, all components involving at least 2 thousand nodes have a duration lower than 10^{-3} seconds, and all components that last for more than two seconds involve less than ten nodes.

More generally, these plots show that there are many strongly connected components with very short duration: 90% last less than 0.14 seconds. These components are due to the *frontier effect*, that we define as follows. Consider a set X of nodes, and assume that link segments that start close to a given time b and end close to a given time e connect them. However, they all start at different times and end at different times. This leads to a connected component $([b', e'], X)$, with b' close to b and e' close to e, but also to many short strongly connected components that both start and end close to b, or close to e. These components make little sense, if any, but they account for a huge fraction of all strongly connected components, and so they have a crucial impact on computation time as explained in the previous section. In Chapter 4, we will introduce an approximation scheme to get rid of such components while keeping crucial information.

3.4 Related Work

First of all, we point out that the stream graph approach differs from the fully-dynamic approach. Here, we focus on the whole history of nodes and links, our objective being to retrieve all connected components that were present at any instant in time. It does not consist in maintaining the current state of a graph after insertions and deletions of links. Regarding connected components in fully-dynamic formalisms we refer to [60] and [15] (see Section 3.2.2 for an adaptation of those methods).

We focus here on connected components defined in [59], but other notions of connected components in dynamic graphs have been proposed. Several of them rely on the notion of reachability, which, in most cases, induces components that may overlap and are NP-hard to enumerate, see for instance [16, 73, 42, 49]. This makes them quite different from the connected components considered here. The fact that they form a partition of temporal nodes, and may be computed in polynomial time and space, are two important features.

Akradi and Spirakis [6] study and propose an algorithm for testing whether a given dynamic graph is connected at all times during a given time interval. If it is not connected, their algorithm looks for large connected components that exist for a long duration. Vernet *et al.* [100] propose an algorithm for computing all sets of nodes that remain connected for a given duration, and that are not dominated (either regarding their size or duration) by others such sets. Unlike our work, these papers do not look for a partition of the set of temporal nodes, as not all temporal nodes belong to a component, and in the case of [100] two components may overlap.

Finally, [23] computes t-interval connectivity in dynamic graphs. It tests whether the intersection of graphs existing during t consecutive time steps is connected, which does not directly lead to a notion of connected components.

The distinction between strongly and weakly connected components is classical for directed graphs. It has in particular been observed that, while there is a giant weakly connected component in the case of the web, the strongly connected components are smaller [20]. Nicosia *et al.* [73] study weakly and strongly connected components in time-varying graphs, with a notion of connectivity based on reachability through temporal paths, which is not symmetric. Therefore they have a component for each node, which may overlap. They study in several real-world datasets how the size of these components evolve when varying the length of the time interval considered.

3.5 Conclusion

We proposed, implemented, and experimentally assessed a family of polynomial algorithms to compute the connected components of stream graphs. These algorithms handle streams of dozens of millions of events, and output all connected components in a streaming fashion. They make connected components usable in practice, as we illustrate on a large-scale real-world dataset. Up to our knowledge, it is the first time that a partition of temporal nodes into connected components is computed at such scales.

Interestingly, our algorithms leave room for heuristics in the order in which events are processed. In the case of UF SCC, for instance, link segments may be considered in any order, and handling long segments first would speed up computations. Likewise, choosing appropriate links for data structure updates in FD SCC may have a strong impact.

Another promising direction is to enumerate connected components without listing them: one may for instance output only component size, duration and span in this way. Fully dynamic algorithms are particularly appealing to this regard, as their complexity is dominated by the explicit component listing. Similarly, one may encode connected components in ways that make listing more compact, and so faster; one may for instance only list component changes, or output components with non-contiguous time spans. These directions call for extended definitions, though.

4 Alternative Stream Graph Representations

Contributions

- A connectivity based data representation of a stream graph which greatly reduces the complexity of computing reachability queries: the condensation of a stream graph
- An alternative data representation paving the way to an efficient parallel framework for the computation of numerous properties on stream graphs: the stable directed acyclic graph of a stream graph
- An approximation method speeding up numerous methods in practice while preserving the connectivity properties of a stream graph: the Δ-approximation

In this chapter we present alternative representations - data structures designed to facilitate specific computations - of stream graphs. The main motivation, as for graph representations, is to be able to perform certain tasks efficiently. In order to do so, we design data structures to address different types of queries or to ease the computation of particular measures and properties.

For instance, in chapter 2 we introduced a compact data structure to represent stream graphs. This representation allows constant time access to nodes and presence times of a given node, and linear time to links and presence times of a given link. However, this representation, as shown in chapter 3, does not allow efficient reachability queries. In the following we propose several structures supporting different kind of queries efficiently - like reachability queries - at the expense of a preprocessing time and a greater memory consumption.

To our knowledge, no other work has addressed alternative representations of temporal graphs with a continuous modeling of the temporal dimension. We can refer to [12] for alternative representations of snapshot sequences. Specifically, data representations for reachability queries in temporal graphs have been addressed in [108], [113] and [85].

Our first stream graph representation, the condensation (Section 4.1), uses strongly connected components as building blocks and reduces the complexity of reachability queries. After defining this representation, we propose a method to compute the condensation from a stream graph, based on an algorithm presented in chapter 3. Then, we use its main properties to design algorithms answering reachability queries and proceed to evaluate them on real world stream graphs. Second, we present the stable directed graph (Section 4.2) and use its properties to design a parallel framework which efficiently handles the computation of numerous stream graphs properties. Third, we propose an approximation method, the Δ-approximation (Section 4.3), preserving a stream graph's connectivity while improving the computation time of several methods. Finally, we conclude by proposing ideas to improve these stream graphs representations and suggesting other potential applications (Section 4.4).

4.1 Condensation

In previous chapters we showed that one can partition a stream graph into weakly or strongly connected components. We now present a method to take advantage of these partitions: we use connectivity properties of these structures to build an object which facilitates the answering of many reachability queries of the form "is (t', v) reachable from (t, u)?" or "is there a path from u to v?"

Reachability queries constitute a major research field in graph theory due to its numerous applications in biology, transportation or traffic networks. Most approaches consist in a labelling scheme based on a particular node ordering, applied to the graph itself or to alternative representations - such as its transitive closure. This labelling scheme allows the creation of indexes reducing computation time of reachability queries. We refer to a survey of these methods [114] and for more recent ones to [115], [116] and [108]. The vast majority of these methods are compatible with a dynamic approach, as the maintained indexes or representations support dynamic updates - addition or deletion of nodes and links. However, they cannot be easily adapted to stream graphs: as shown in chapter 1, a stream graph contains the whole history of a dynamic graph. A query may involve elements with different temporalities, therefore an index at a specific instant in time cannot answer these kinds of queries. In this section we introduce an object supporting numerous reachability queries and compatible with state-of-the-art indexing methods: the **condensation** of a stream graph.

The condensation of a directed graph G is the directed acyclic graph where each strongly connected component is contracted to a single vertex.

Our approach is similar to the ones presented in [113] and [85]. We argue that our method generalises the concept of condensation graph [5] by taking into account a continuous temporal dimension. For consistency, we choose to keep the same name.

4.1.1 Definitions

Definition 4.1.1. *We say that a cluster, defined in 1.2.5, $C' = (I', X') \in W$ **follows** another cluster $C = (I, X) \in W$ if they satisfy the three following conditions:*

- $\forall t \in I, \forall t' \in I',\ t < t'$.
- $\overline{I} \cap \overline{I'} \neq \emptyset$ *where* $\overline{I}$ *denote the topological closure of* I.
- $X \cap X' \neq \emptyset$.

Less formally, a cluster follows another if we can construct a path of zero length and duration from the first to the second. They are adjacent in the graphical representation, for example in Figure 4-1 cluster 2 follows cluster 1.

Using this definition, we construct the *condensation* of S. Let us recall that the set of strongly connected components, denoted by $\mathscr{C}$, of a stream graph, is a partition of the set of temporal nodes W (cf Chapter 3).

Definition 4.1.2. *We define $G_{\mathscr{C}} = (\mathscr{C}, E_{\mathscr{C}})$, the **condensation** of S as the graph with node set $\mathscr{C}$, in which (C, C') is in $E_{\mathscr{C}}$ if and only if C' follows C. We denote by $n_c = |\mathscr{C}|$ and $m_c = |E_{\mathscr{C}}|$ its number of nodes and (directed) links, respectively.*

We say that (t, v) is in $C = (I, X)$ if $t \in I$ and $v \in X$. For any (t, v) in W, we denote by $C(t, v)$ the unique strongly connected component $C \in \mathscr{C}$ such that (t, v) is in C and by $C(v) = \{(I, X) \in \mathscr{C}$ such that $v \in X\}$ the set of strongly connected components containing v.

The upper part of Figure 4-1 represents partition into strongly connected components - condensation nodes - of the stream graph of Figure 1-1. The lower part of Figure 4-1 shows the condensation graph of the stream. In the following we will often consider nodes of the condensation graph as "components" in order to facilitate the description of some methods.

Proposition 4.1.1. *The condensation $G_{\mathscr{C}}$ of a stream graph S is a Directed Acyclic Graph.*

Proof. One can easily notice that the presence of a cycle in the condensation graph means that at least one directed link infringes the first condition of definition 4.1.1. □

Proposition 4.1.2. *The condensation $G_{\mathscr{C}}$ of a stream graph S has the following properties:*

1. *The number of nodes of $G_{\mathscr{C}}$, n_c, is in $O(N + M)$.*
2. *The number of links of $G_{\mathscr{C}}$, m_c, is in $O(M)$.*

Proof. 1. The number of nodes of $G_{\mathscr{C}}$ is the number of strongly connected components of the stream graph. From proposition 3.2.1, it is in $O(N + M)$.

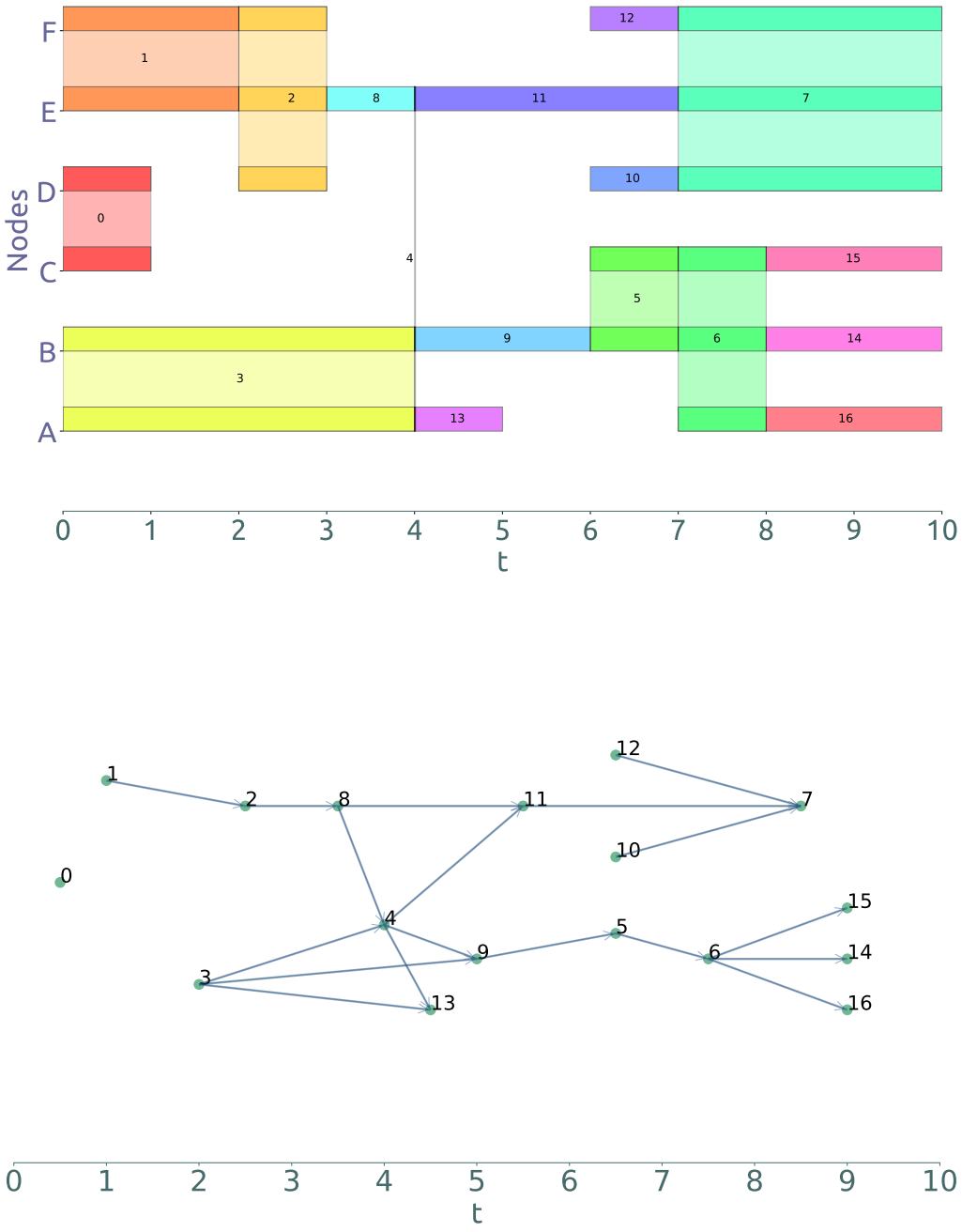

Figure 4-1: The 17 strongly connected components of the stream graph of Figure 1-1 (top) and the corresponding Condensation Directed Acyclic Graph (bottom).

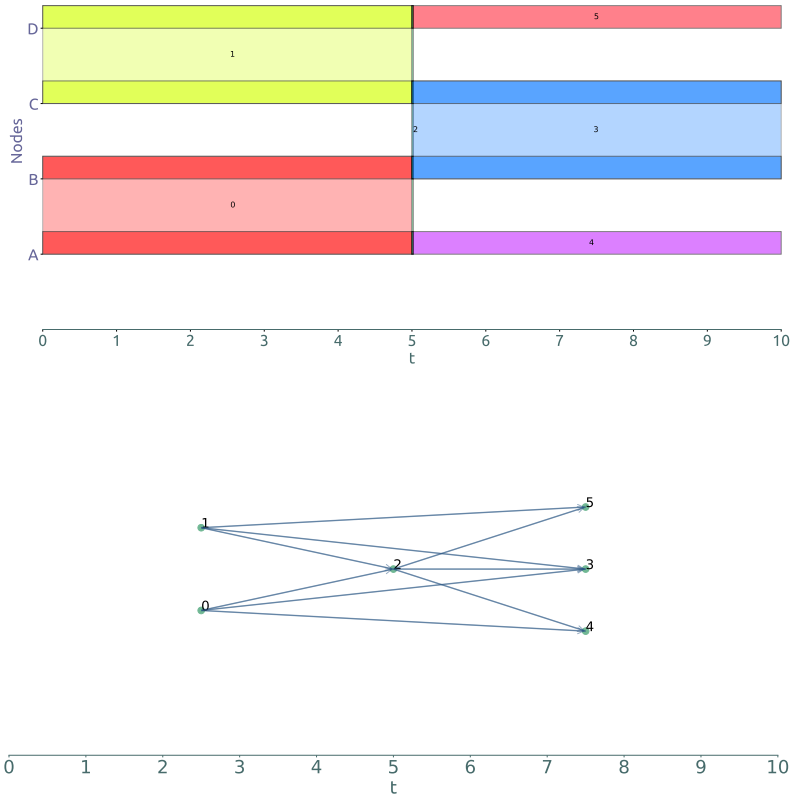

Figure 4-2: Illustration of the impact of a link's beginning on condensation links.

2. A link in $G_{\mathscr{C}}$ corresponds to a following relationship between two strongly connected components. Each link segment can induce up to seven links in $G_{\mathscr{C}}$. Indeed the beginning of a link segment may correspond to the beginning of two components: one instantaneous at the link segment beginning and one starting just after. As the link segment may connect nodes in two different components before its beginning, it could lead to five links in $G_{\mathscr{C}}$: a link between the created components as they can follow each other and four links from the previous components to the created ones, as they can also have a following relationship. The ending of a link segment may correspond to the beginning of two connected components, if the current component becomes disconnected, leading to two other links in $G_{\mathscr{C}}$.

   For instance, in Figure 4-2, the beginning of the link $([5, 10], BC)$ leads to the creation of two components (2 and 3) and to the condensation links $(1, 2)$, $(1, 3)$, $(0, 2)$, $(0, 3)$ and $(2, 3)$. The ending of links $([0, 5], AB)$ and $([0, 5], CD)$ creates the condensation links $(0, 4)$ and $(1, 5)$ as well as other links already mentioned.

□

4.1.2 Algorithm

In Chapter 3 we proposed several algorithms to compute strongly connected components, the best one being *SCC Direct*. It can be adapted to output the condensation of a stream graph and we call it *SCC-Condensation Direct*.

We can notice that for a component to follow another it must be consecutive to either a link arrival or a link departure. A node arrival creates a new component which cannot follow another one, as it does not share any node with any other. A node departure ends a component without giving birth to another, discarding any possibility for a condensation link.

SCC-Condensation Direct is derived from *SCC Direct* in the following way, during step 2 (link arrival) of *SCC Direct* 3.2.1, if components are outputted, we create condensation links from the outputted components - $C_u = (\langle b_u, t[, X_u)$ and $C_v = (\langle b_v, t[, X_v)$ - to the created one - $C_{uv} = ([t, X_u \cup X_v)$. Likewise, during step 3 (link departure), if the removal causes components to become disconnected we create condensation links from the initial component - $C_u = (\langle v_u, t], X_u)$ - to the disconnected ones - C'_u and C'_v.

Proposition 4.1.3. *Algorithm SCC-Condensation Direct computes the condensation of a stream graph in $O(M \cdot (m+n) + N)$ time and $O((N+M) \cdot n)$ space.*

Proof. Additional steps are in $O(1)$ and do not impact the time complexity of *SCC Direct* (see 3.2.3). By proposition 4.1.2, a condensation has at most M links and $N + M$ components (condensation nodes) and each of these components contains at most n nodes. Hence, a space complexity in $O((N+M) \cdot n)$. □

Heuristic 4.1.1. *One can observe that number of links in the condensation can be reduced without impacting its connectivity. We can obtain a transitive reduction of the condensation by removing links $(u, w) \in E_{\mathscr{C}}$ such that $\exists (u, v)$ and $(v, w) \in E_{\mathscr{C}}$.*

For instance, in Figure 4-2, we can notice that links $(1, 5)$, $(1, 3)$, $(0, 4)$ and $(0, 3)$ can safely be removed without impacting the connectivity.

To compute this transitive reduction, we build the adjacency list of $G_{\mathscr{C}}$: $A_{\mathscr{C}}$ - each node is associated to a set containing its neighbors. Then, for each node $C \in \mathscr{C}$ we consider every pair (C', C'') such that $C', C'' \in A_{\mathscr{C}}[C]$. If C' is in $A_{\mathscr{C}}[C'']$ we can safely remove (C, C') from $E_{\mathscr{C}}$. And if C'' is in $A_{\mathscr{C}}[C']$ we can safely remove (C, C'') from $E_{\mathscr{C}}$.

The time complexity of the above heuristic is in $O(\sum_{u \in \mathscr{C}} d_{out}(u)^2)$ where $d_{out}(u)$ is the out-degree of $u \in \mathscr{C}$ in $G_{\mathscr{C}}$. We can express this complexity according to n_c by posing $\alpha = \frac{\sum_{u \in \mathscr{C}} d_{out}(u)^2}{n_c}$, the complexity of the above heuristic becomes $O(\alpha \cdot n_c)$. In Section 4.1.5 we will show that, in practice, α has a small value and that this heuristic is linear in n_c making the cost of this heuristic negligible compared to the one of *SCC-Condensation Direct* (see Table 4.1).

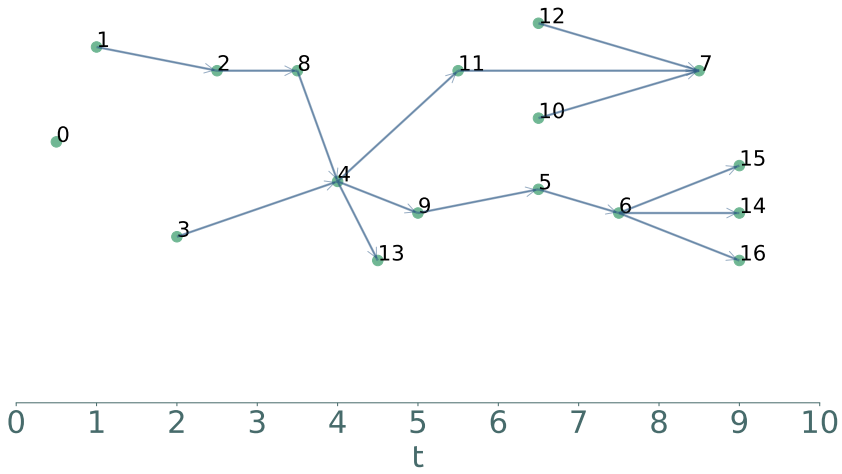

Figure 4-3: Relaxed Condensation Directed Acyclic Graph corresponding to the stream graph of Figure 1-1

Figure 4-3 shows the relaxed condensation graph of the stream graph of Figure 1-1. Links $(3, 9)$, $(3, 13)$ and $(8, 11)$ were removed. However, this relaxation does not modify the asymptotic number of condensation links.

4.1.3 Connectivity Properties

The main property of the condensation of a stream graph is the following: any path in S corresponds to a path in $G_{\mathscr{C}}$. We provide an example of this property in Figure 4-4 and we demonstrate this property below.

Lemma 4.1.1. *If there is a path from (t, u) to (t', v) in S then either $C(t, u) = C(t', v)$, or there is a path from $C(t, u)$ to $C(t', v)$ in $G_{\mathscr{C}}$.*

Proof. If (t, u) and (t', v) belong to the same strongly connected component we have $C(t, u) = C(t', v)$. Otherwise let $P = (t_0, u_0, v_0), \ldots, (t_k, u_k, v_k)$ be the path between $(t_0, u_0) = (t, u)$ and $(t_k, v_k) = (t', v)$. We can define a sequence $(C_i)_{i\in\{0,k\}} \in \mathscr{C}$ such that $(t_0, u_0), (t_0, v_0) \in C_0, \ldots, (t_k, u_k), (t_k, v_k) \in C_k$. Then let us suppose that there exists $j \in \{0, k-1\}$ such that $C_j \neq C_{j+1}$ and $C_{j+1} = (I_{j+1}, X_{j+1})$ doesn't follow $C_j = (I_j, X_j)$ in $G_{\mathscr{C}}$. Using the properties of P we have $t_j < t_{j+1}$ and $v_j = u_{j+1}$ which means that v_j is present from t_j to t_{j+1}, thus $X_j \cap X_{j+1} \neq \emptyset$. We can define $h \in T$ such that $t_j < h < t_{j+1}$ where $(h^-, v_j) \in C_j$ and $(h^+, v_j) \in C_{j+1}$, hence $\overline{I_j} \cap \overline{I_{j+1}} \neq \emptyset$. Using the fact that C_j and C_{j+1} are maximal we also have $\forall t, t' \in I_j, I_{j+1}, t < t'$. By construction of $G_{\mathscr{C}}$: C_{j+1} must follow C_j which is a contradiction. □

Lemma 4.1.2. *If there is a path from C to C' in $G_{\mathscr{C}}$ and $C \neq C'$ then for all (t,u) in C and (t',v) in C' there is a path from (t,u) to (t',v) in S.*

Proof. Suppose that there is a path $P = (C_0, C_1, \ldots, C_k)$ from $C_0 = C$ to $C_k = C'$ in $G_{\mathscr{C}}$ and that there exists $(t,u) \in C$ and $(t',v) \in C'$ such that we cannot reach (t',v) from (t,u) in S. Using the fact that $C_0 = (I_0, X_0)$ is a SCC we can define a path from (t,u) to any node $v_0 \in C_0$. We choose this element v_0 to be in $X_0 \cap X_1$, which is not empty given the fact that $C_1 = (I_1, X_1)$ follows C_0. We also choose an instant $t_1 \in I_1$, a node $u_1 \in X_1$ and we define the path $P_0 = (t, u, v_0), (t_1, v_0, u_1)$. We can iterate this process for any couple (C_i, C_{i+1}), $i \in \{0, k-1\}$ of components in P in order to obtain a sequence $(P_i)_{i \in \{0,k-1\}}$. If we concatenate this sequence we obtain a path going from (t,u) to (t',v) which is a contradiction. □

These lemmas show that the condensation of a stream graph encodes much information about paths and reachability. Indeed all paths corresponds to a sequence of elements of $\mathscr{C}$, in other words, any path in S may be divided into a sequence of slices, each consisting in a path in - the substream induced by - a strongly connected component of S and so a node of $\mathscr{C}$. This leads to the following theorem:

Theorem 4.1.1 (path slicing)**.** *Let us consider a stream graph $S = (T, V, W, E)$, and two temporal nodes (i,u) and (j,v) in W. There is a path $P = (t_0, u_0 v_0), (t_1, u_1 v_1), \ldots, (t_k, u_k v_k)$ from (i,u) to (j,v) in S if and only if there is a path $P_{\mathscr{C}} = C_0, C_1, \ldots, C_l$ in $G_{\mathscr{C}}$ such that there exists a sequence $x_0, x_1, \ldots, x_l$ s.t. $(t_0, u_0 v_0), \ldots, (t_{x_0}, u_{x_0} v_{x_0})$ is a path in $S(C_0)$, $(t_{x_0+1}, u_{x_0+1} v_{x_0+1}), \ldots, (t_{x_1}, u_{x_1} v_{x_1})$ is a path in $S(C_1)$, and so on until $(t_{x_{l-1}+1}, u_{x_{l-1}+1} v_{x_{l-1}} + 1), \ldots, (t_{x_l}, u_{x_l} v_{x_l})$ is a path in $S(C_l)$.*

Proof. This theorem comes from lemmas 4.1.1 and 4.1.2. □

The main consequence of this theorem is that we can encode a potentially infinite set of paths between two temporal nodes in S by a finite set of distinct paths in $G_{\mathscr{C}}$ (where each path is a finite sequence of components). A path in S is denoted by P_S and its representation in $G_{\mathscr{C}}$ by $P_{\mathscr{C}}$. In the following we will often speak in terms of **stream graph path** and **condensation path** in order to avoid any confusion. Consequently, reachability queries - asserting the existence of a path in S - can be answered by browsing $G_{\mathscr{C}}$.

Remark 4.1.1. *We can easily extract an arbitrary stream path from a condensation path $P_{\mathscr{C}} = (C_0, C_1, \ldots, C_k)$. Each step in $P_{\mathscr{C}}$ corresponds to a transition from a $C = (I, X)$ to a $C' = (I', X')$, with $(C, C') \in E_{\mathscr{C}}$, in order to keep the coherence of the path, it must pass through a node belonging to $X \cap X'$. For all $i \in \{0, k\}$ we can construct a subpath P_i inside $S(C_i)$ where $C_i = (I_i, X_i) \in P_{\mathscr{C}}$: P_i starts from a node in $X_{i-1} \cap X_i$ and reaches a node in $X_i \cap X_{i+1}$. The concatenation of these P_i results in a stream path.*

The worst-case complexity of this extraction procedure is in $O(|P_{\mathscr{C}}|(n+m)) = O(n_c(n+m))$, as lengths of paths in $G_{\mathscr{C}}$ - an acyclic directed graph - cannot exceed n_c. Since

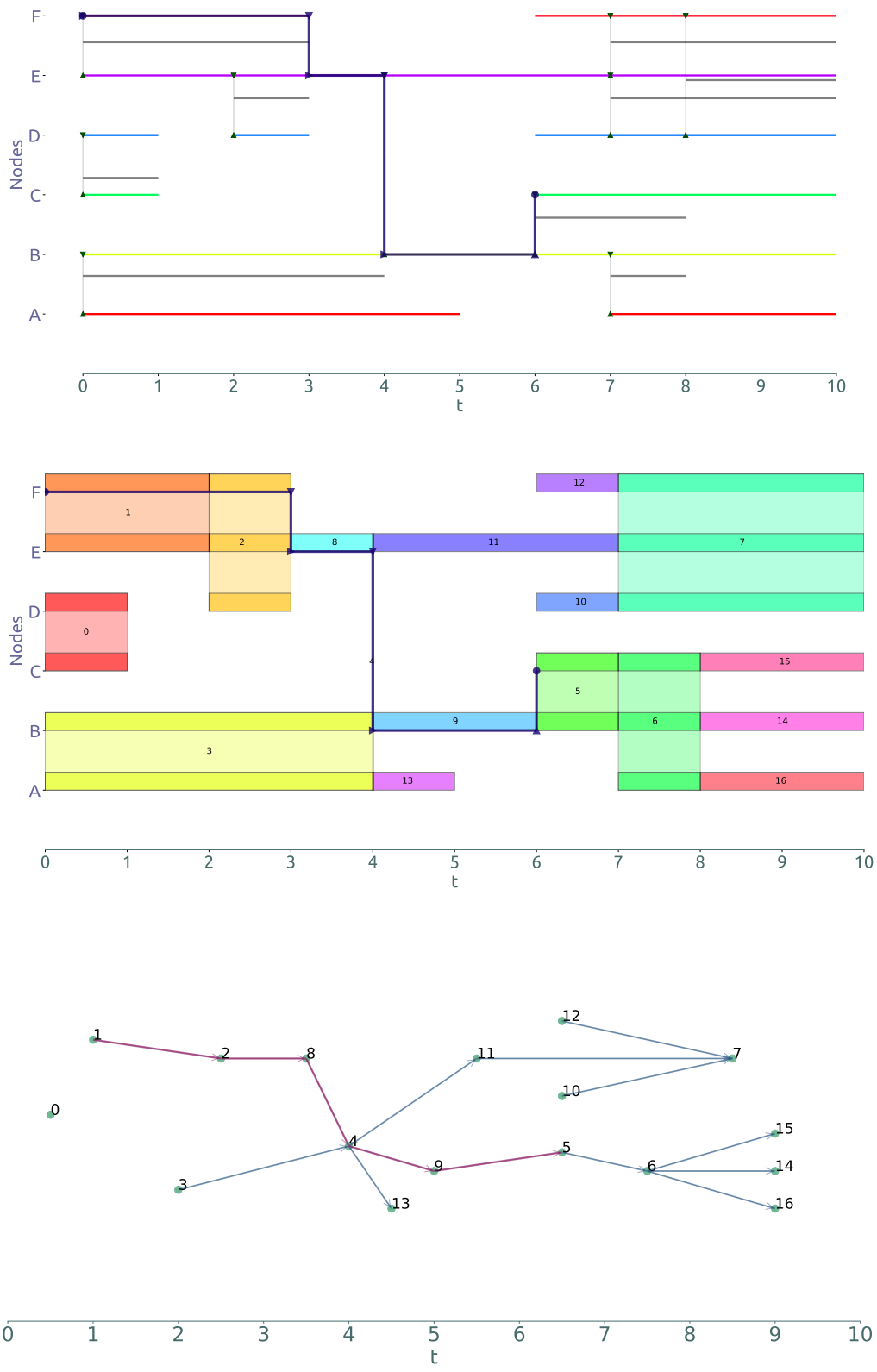

Figure 4-4: A temporal path from $(0, F)$ to $(6, C)$ in the stream graph (top and middle) and the equivalent path in its condensation $((1, 2, 8, 4, 9, 5))$ (bottom).

a component in $\mathscr{C}$ may contain, at most, n nodes and m links, finding a path in such a component may require $O(n+m)$.

4.1.4 Reachability Queries

As discussed in chapter 3, a stream graph can be partitioned into distinct weakly connected components. We can create a distinct condensation for each WCC. An index associating each node segment to its WCC can be created during *WCC DFS* (see Section 3.1) in $O(N)$ time and $O(N)$ space. Reachability queries between elements of different WCC can be answered in $O(1)$. In the following, without a loss of generality, we assume that queries involve elements of the same WCC.

Given two temporal nodes (t,u) and $(t',v) \in W$ and two nodes w and $z \in V$, there can be four types of reachability queries and each one has its equivalent in $G_{\mathscr{C}}$:

1. Is there a path from (t,u) to (t',v) in $S \iff$ is there a path from $C(t,u)$ to $C(t',v)$ in $G_{\mathscr{C}}$.

2. Is there a path from (t,u) to z in $S \iff$ is there a path from $C(t,u)$ to $C(z)$ in $G_{\mathscr{C}}$

3. Is there a path from w to (t',v) in $S \iff$ is there a path from $C(w)$ to $C(t',v)$ in $G_{\mathscr{C}}$

4. Is there a path from w to z in $S \iff$ is there a path from $C(w)$ to $C(z)$ in $G_{\mathscr{C}}$

Queries 1 and 2 can be answered using a single breadth first search on $G_{\mathscr{C}}$, in $O(n_c + m_c)$ time.

Queries 3 and 4 are more complex as $C(w)$ and $C(v)$ may contain several nodes of $G_{\mathscr{C}}$. We address them in chapter 5 and show that they can also be answered in $O(n_c+m_c)$.

We point out that for these querying algorithms to be efficient the condensation should be stored in main memory as the time complexity of reading a stream graph's condensation exceeds the one of these algorithms.

The data structure used in Straph to encode the condensation of a stream graph has been detailed in chapter 2.
In Straph, the computation of the condensation of a stream graph as well as reachability queries is very simple. We give a practical example below.

```
S = stream_graph()
# Computation of the condensation of the Stream graph
cdag = S.condensation_dag()
t1, t2, u, v = 0, 10, 'A', 'B'
# The function 'is_reachable' returns a Boolean
# Node to Node
cdag.is_reachable(source = u, target = v)
# Temporal Node to Temporal Node
cdag.is_reachable(source= (t1,u),target = (t2,v))
# Node to Temporal Node
cdag.is_reachable(source= u,target = (t2,v))
# Temporal Node to  Node
cdag.is_reachable(source= (t1,u),target = v)
```

The condensation being a directed acyclic graph, it is possible to significantly improve reachability queries by applying an indexing method on it such as *TopChain* [108] or *TOL* [116]. We hope to evaluate the benefits of these methods in the future as well as providing them in Straph.

4.1.5 Experiments

Structural properties of $G_{\mathscr{C}}$ can be decisive, as complexity parameters. We have evaluated them on twelve real world datasets, described in Section 1.3: *UC Message (UC)*, *High School 2012 (HS 2012)*, *Digg*, *Infectious*, *Twitter Higs (Twitter)*, *Linux Kernel mailing list (Linux)*, *Facebook wall posts (Facebook)*, *Epinions*, *Amazon*, *Youtube*, *Movielens* and *Wiki Talk En (Wiki)*.

As mentioned previously, to efficiently answer reachability queries, a stream graph condensation should be stored in main memory. Therefore, as the space complexity of the condensation is in $O((M+N)\cdot n)$, some stream graphs exceed our main memory, such as in *Mawilab 2020-03-09 (Mawilab)* and *Stackoverflow* datasets. Consequently we ignore these two datasets in this section. However, as shown in chapter 3, it is possible to design a streaming algorithm and output a stream graph's condensation in a streaming fashion. In section 4.3, we propose an approximation method reducing the number of strongly connected components and tackling this issue.

First of all, we notice that in *Epinions* and *Amazon* datasets there is no link in the condensation. This observation can be explained by the fact that each strongly connected component of these datasets is also a weakly connected component (see Table 4.1). Therefore, no SCC is adjacent to another - and no condensation link exists - as they would belong to the same weakly connected component otherwise. It could be argued that the δ value we used for the temporal links' minimal duration is too short regarding the duration between two logged timestamps.

Real world condensation graphs are very sparse ($m_c < n_c$), as we can observe in Table 4.1, the mean out-degree of $G_{\mathscr{C}}$, d_c, is very low (< 1). The values of α are also low, unless for the *youtube* dataset, making heuristic 4.1.1 efficient in practice. Moreover, as observed in Figure 4-5, $n_c \sim N$ and $m_c \sim M$. Nevertheless the size of the

	Running Time (s)	n_c	m_c	d_c	α	$\lvert\mathscr{W}\rvert$
UC	1.130	53 940	43 273	0.802243	0.907	11 276
HS 2012	1.412	50 312	39 060	0.776356	0.993	13 253
Digg	3.400	143 953	118 953	0.826332	0.963	26 078
Infectious	2.543	106 306	95 822	0.901379	1.179	16 990
Twitter	124.879	579 683	472 097	0.814405	0.881	113 200
Linux	32.943	697 704	658 207	0.94339	1.118	62 918
Facebook	25.779	794 986	422 273	0.53117	0.574	373 217
Epinions	6.562	75 648	0	0	0	75 648
Amazon	145.337	4 092 400	0	0	0	4 092 400
Youtube	2970.871	1 168 736	503 666	0.430949	142.286	738 638
Movielens	2212.060	12 480 067	12 214 848	0.978749	1.002	282 043
Wiki	3010.126	23 516 709	19 439 127	0.826609	0.892	4 206 816

Table 4.1: Running time in seconds of algorithm *SCC-Condensation Direct* - Characteristics of the condensations of real world stream graphs ($n_c = |\mathscr{C}|$) the number of nodes (and of SCC), m_c the number of links, d_c the mean out degree) - $\alpha = \frac{\sum_{u\in\mathscr{C}} d_{out}(u)^2}{n_c}$ the complexity parameter in Heuristic 4.1.1 - $|\mathscr{W}|$ the number of weakly connected components.

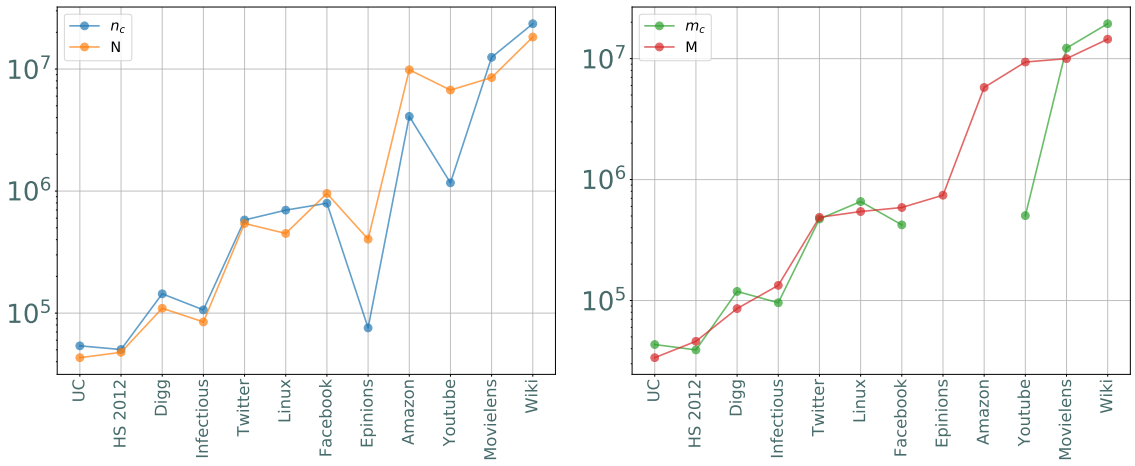

Figure 4-5: Number of nodes in $G_{\mathscr{C}}$, n_c, along with the number of node segments in S, N, (left) and number of links in $G_{\mathscr{C}}$, m_c, along with the number of link segments in S, M, (right) for each considered real world stream graph (horizontal axis, ordered with respect to M).

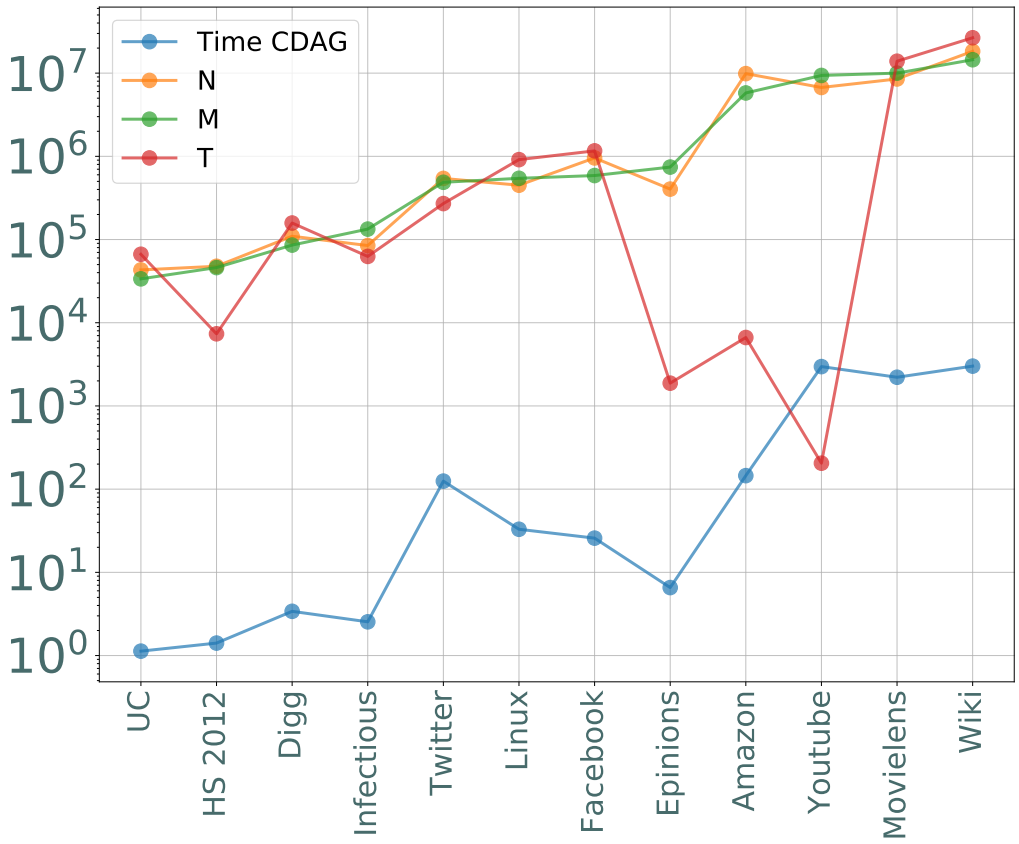

Figure 4-6: Running time of *SCC-Condensation Direct* in seconds along with the number of link segments, M, node segments, N, and event times, T, in S for each considered real world stream graph (horizontal axis, ordered with respect to M).

condensation graph is not necessarily lower than the one of the stream graph it stems from. In our datasets, the size of $G_{\mathscr{C}}$ heavily depends on the nature of interactions: if they happen at the same time instant or if they begin at close but distinct time instants. In the first case this leads to bigger SCC resulting in lower n_c and m_c. In the second case this results in multiple and successive component insertions leading to higher n_c and m_c.
These results will prove to be critical in the design of some algorithms (see chapter 5).

The running time of *SCC-Condensation Direct*, presented in Table 4.1 and Figure 4-6, is bigger than the one of *SCC Direct* due to the fact that we have to build an object for each SCC whereas in *SCC Direct* we output the component as soon as computed. The running times of both algorithms are still correlated to the number of temporal links in the stream graph (see Section 3.3 for more details).

4.2 Stable Directed Acyclic Graph

The main motivation for the following concepts is to decompose a stream graph into building blocks in which the dynamic does not play a part. A graph can be decomposed into connected components that can be analyzed independently, which allows parallel implementations of numerous algorithms; stream graphs lack an equivalent counterpart. As shown in chapter 3, weakly and strongly connected still possess an internal dynamic. In this section we propose a notion of *stable connected component* which can be seen as the connected component of a static graph and that is, consequently, independently analyzable.

4.2.1 Definitions

Definition 4.2.1. *A **stable connected component** of a Stream Graph $S = (T, V, W, E)$ is a maximal cluster $C = (I, X) \subseteq W$ such that $\forall t \in I$, X is a connected component of G_t and $\forall t' \in I$ we have $G_t = G'_t$. The set of stable connected components of a stream graph is denoted by $\mathscr{S}$.*

In other words, a stable connected component is a cluster $C = (I, X)$, $I = [b, e]$ where interactions between the nodes have begun before b or at the same time and have ended after e or at the same time.

It is trivial to show that a stable connected component is also a subset of a strongly connected component. The decomposition into stable connected components is a finer grain decomposition of the stream graph than the one into strongly connected components.

A stable connected component, $C = (I, X)$, can be reduced to a static graph $G_C = (X, E_C)$ spanning I. In practice, interactions occurring inside a stable connected component can be considered as static links. The corresponding object only consists of a time window, I, and an adjacency list, A_C.

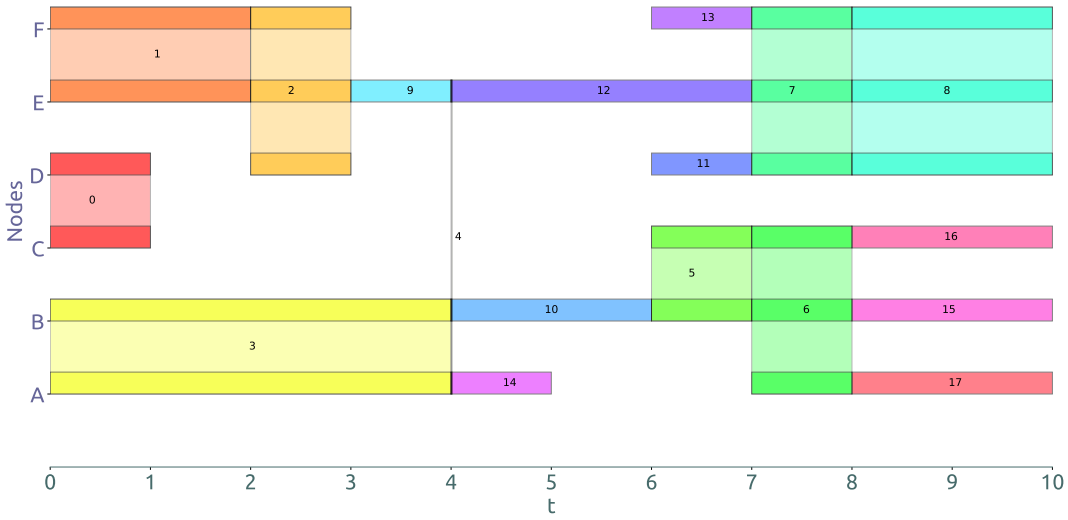

Figure 4-7: Stable connected components of the stream graph of Figure 1-1.

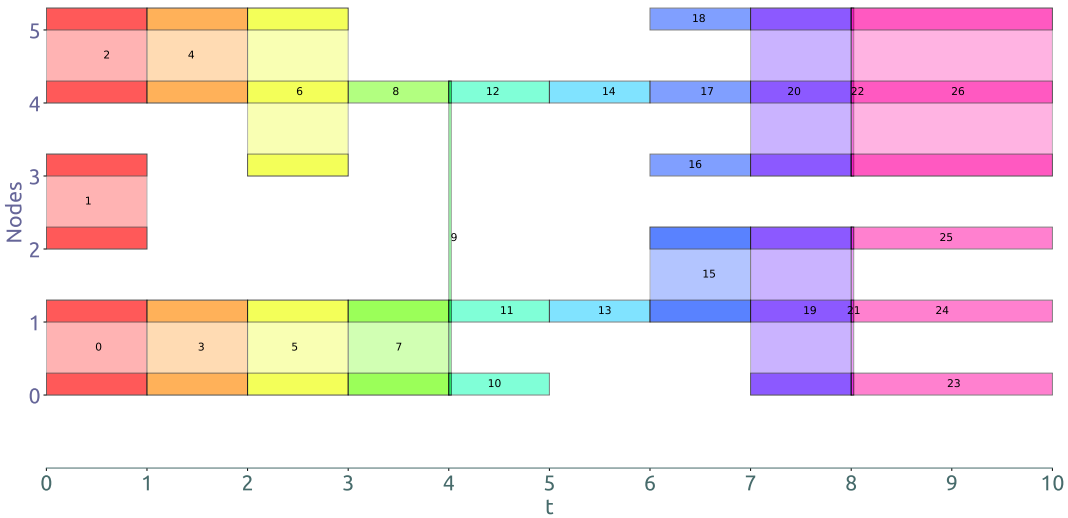

Figure 4-8: Illustration of the decomposition into snapshots. Colors indicate distinct snapshots and numbers indicate connected components in snapshots.

This decomposition is different from the one into distinct snapshots. Indeed the decomposition into snapshots is only "vertical": we take slices of the stream graph along the time dimension. The decomposition into stable connected components is both "vertical" and "horizontal", we obtain "rectangles" of different heights (X) and lengths (I). This difference is illustrated in Figures 4-7 and 4-8: we either obtain 27 connected components in 11 distinct snapshots or 18 stable connected components. In huge real-world datasets, this difference becomes very important and can be critical from an algorithmic perspective (see Table 4.2).

Definition 4.2.2. *We define $G_{\mathscr{S}} = (\mathscr{S}, E_{\mathscr{S}})$, the **stable directed acyclic graph** of S as the graph with node set $\mathscr{S}$, in which (C, C') is in $E_{\mathscr{S}}$ if C' follows C. We denote by $n_s = |\mathscr{S}|$ and $m_s = |E_{\mathscr{S}}|$ its number of nodes and (directed) links, respectively.*

Propositions 4.1.1 and 4.1.2 also hold for the stable DAG of a stream graph.

4.2.2 Algorithm

To obtain the stable DAG of a stream graph we can adapt the algorithm *SCC Direct* proposed in chapter 3 to compute strongly connected components. We call this adaptation *SCC-Stable Direct*.

First notice that a link arrival or a link departure necessarily leads to the creation of a new stable connected component. During step 2 (link arrival) of *SCC Direct* 3.2.1, we create stable links from the outputted components to the created one. Likewise, during step 3 (link departure), if the removal causes components to become disconnected we create stable links from the initial component to the disconnected ones but if the component remains connected we also create a new component and a stable link between the two. This method can be incorporated as a subroutine of *SCC Direct* without altering its time and space complexities.

Proposition 4.2.1. *Algorithm SCC-Stable Direct computes the stable DAG of a stream graph in $O(M \cdot (m+n) + N)$ time and $O((N+M) \cdot n)$ space.*

The number of links in the stable DAG can also be decreased by the application of Heuristic 4.1.1.

If we want to obtain stable connected components directly from strongly connected components, we proceed as follows. Let $C = (I, X) \in \mathscr{C}$, $I = [b, e]$ and $S(C) = (I, X, C, E_C)$ the substream induced by C, for each event time corresponding to the arrival or departure of a temporal link (t, u, v) in E_C we "slice" C into $C' = (I', X)$ with $I' = [b, t]$ and $C'' = (I'', X)$ with $I'' = [t, e]$. Once every event time has been browsed, sliced components are the stable connected components. The addition of stable links between sliced component is trivial. We can notice that this procedure is also highly parallelisable.

In Straph, the stable connected components and the stable DAG inherit from the same objects as the strongly connected components and the condensation. Below, we

	Time SDAG	n_s	m_s	d_s	CC in snapshots
UC	1.204	56 361	45 694	0.810	756 518
HS 2012	1.221	56 166	44 914	0.799	350 726
Digg	3.459	146 268	121 268	0.829	22 886 293
Infectious	3.093	139 387	128 903	0.925	415 640
Twitter	134.186	589 652	482 066	0.817	126 088 469
Linux	42.062	875 167	835 670	0.955	19 645 948
Facebook	27.423	802 262	429 549	0.535	306 150 715
Epinions	6.494	75 648	0	0	75 648
Amazon	134.280	4 092 400	0	0	4 092 400
Youtube	2908.572	1 169 749	504 679	0.431	2 701 203
Movielens	2575.792	13 998 249	13 733 030	0.981	72 509 577
Wiki	3133.755	24 060 565	19 982 983	0.831	2 213 627 558

Table 4.2: Running time in seconds of algorithm *SCC-Stable Direct* - Characteristics of the stable DAG of real world stream graphs (n_s the number of nodes, m_s the number of links,d_s the mean out-degree) - Total number of connected components in the whole sequence of snapshots

provide a snippet of code to obtain these objects in practice.

```
S = stream_graph()
# Computation of the stable connected components of S:
stable_cc = S.stable_connected_components()
# Computation of the stable dag of S:
sdag = S.stable_dag()
```

4.2.3 Experiments

Sizes of real-world stable DAG are presented in Table 4.2. We observe that stable DAG are very sparse ($m_s < n_s$). In two datasets, *Epinions* and *Amazon*, there is no link in their stable DAG. The reason is the same as for their condensation (see section 4.1). Figure 4-9 shows that $n_s \sim N$ and $m_s \sim M$. These results are similar to the ones observed for their condensations.
We notice, in Figure 4-11, that condensations and stable DAG of real-world stream graphs have almost the same number of nodes and links. This means that **strongly connected components are, most of the time, also stable connected components**. The internal dynamic in SCC is very low and the beginning or ending of a temporal link often breaks the strong connectivity.

The running times of *SCC-Condensation Direct* and *SCC-Stable Direct*, presented in Figure 4-10, are nearly identical, as expected.

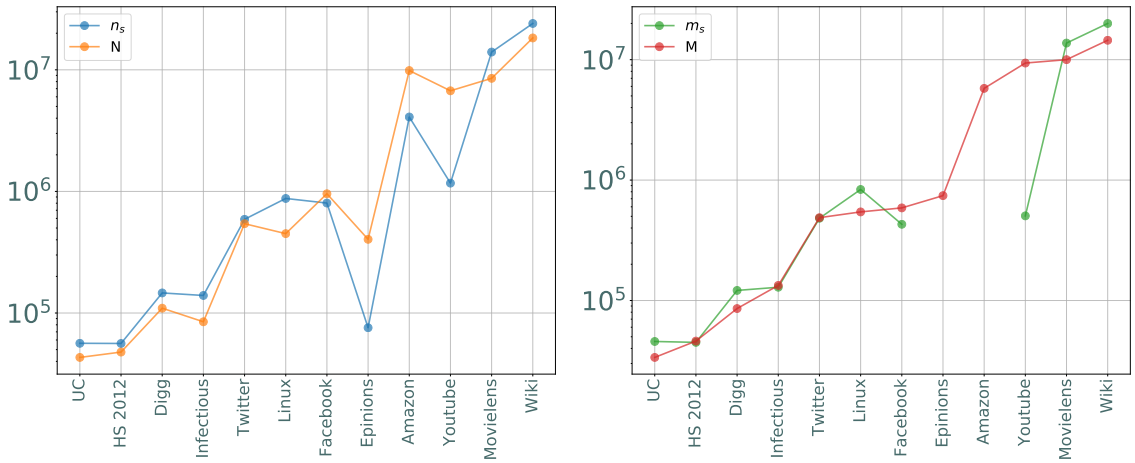

Figure 4-9: Number of nodes in $G_{\mathscr{S}}$, n_s along with the number of node segments in S, N (left) and number of links in $G_{\mathscr{S}}$, m_s along with the number of link segments in S, M (right) for each considered real world stream graph (horizontal axis, ordered with respect to M).

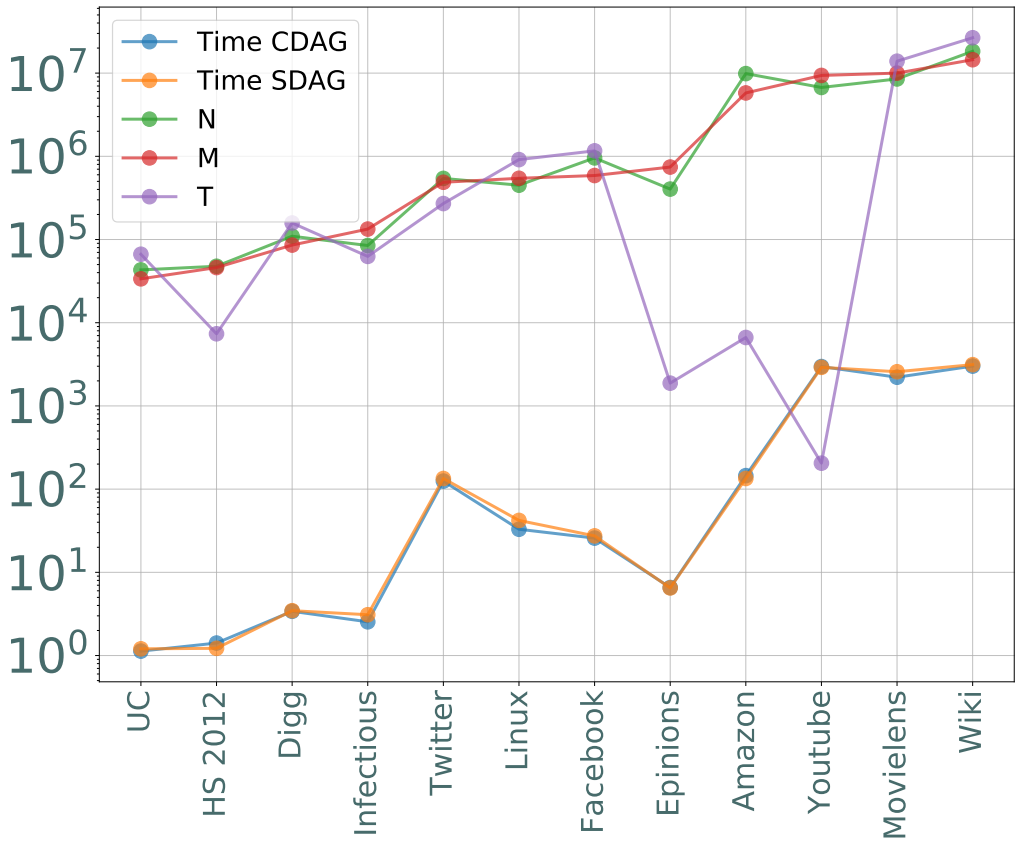

Figure 4-10: Running Time of *SCC-Condensation Direct* and *SCC-Stable Direct* in seconds along with the number of link segments M, node segments N and event times T in S for each considered real world stream graph (horizontal axis, ordered with respect to M).

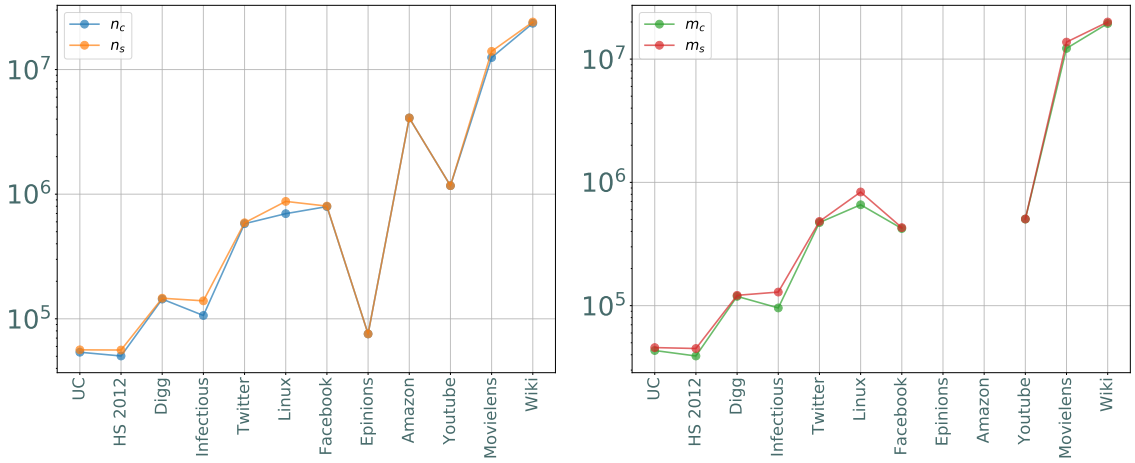

Figure 4-11: Number of nodes in $G_{\mathscr{C}}$, n_c, and in $G_{\mathscr{S}}$, n_s (left) - Number of links in $G_{\mathscr{C}}$, m_c, and in $G_{\mathscr{S}}$, m_s (right) for each considered real world stream graph (horizontal axis, ordered with respect to M).

4.2.4 DAG Parallel Framework

We have shown that a stream graph can be partitioned into stable connected components and that a stable connected component can be considered as a static graph spanning a certain time window.
In this section we present a general method to compute stream graphs properties, using previous results. Let us consider a graph property, for example a node's core number, which is an integer. We will focus on its equivalent in a stream graph: a time series. Consider $\mathcal{P}$ a node (or edge) property in graph theory, given a graph $G = (V, E)$, $\mathcal{P}$ is the following application $\mathcal{P} : V \to \mathbb{R}$ (or $\mathcal{P} : E \to \mathbb{R}$). We recall that the induced static graph of S at time t is denoted by $G_t = (V_t, E_t)$. The equivalent of such a property in a stream graph S is the time series $(\mathcal{P}(V_t))_{t\in T}$ (or $(\mathcal{P}(E_t))_{t\in T}$).

If there exists an algorithm to compute a graph property, this property can be computed in a stream graph, using the following naive method. The stream is divided into distinct snapshots. Then the graph algorithm can be applied, in a parallel fashion, to the connected components of these snapshots. By aggregating consecutive results, we obtain the time series corresponding to the stream graph property.
Previously, we have shown that there can be many more connected components in snapshots than stable connected components resulting in a prohibitive computation time. To solve this problem, we propose a parallel framework based on notions of connectedness to efficiently reuse state-of-the-art static algorithms already implemented in numerous graph libraries, such as NetworkX [43] or NetworKit [93].
Our framework consists in applying an efficient graph algorithm to each stable connected component, represented by an edge list or by an adjacency list/matrix. Each result will be associated to a time period corresponding to the time window of the component. Then, results of adjacent stable connected components may need to be merged together. It can be done by browsing the DAG and aggregated results along

the time axis. This framework is highly parallelisable, as graph algorithms can be computed on each stable connected component independently.

Proposition 4.2.2. *Given a stream graph S and a graph property algorithm of complexity $O(f(n+m))$, the DAG Parallel Framework computes the equivalent stream graph property in $G_{\mathbb{S}}$ in $O((M+N)f(n+m))$.*

Proof. As proposition 4.1.2 is also valid for the stable DAG, we have: $m_s = O(M)$ and $n_s = O(N+M)$. Thus applying an algorithm with complexity $O(f(n+m))$ on each stable connected component can be done in $O(f(n+m)(N+M))$. Aggregating two adjacent results only consists in concatenating intervals, time periods. The whole aggregation step can be done by browsing $G_{\mathscr{S}}$, necessitating $O(M+N)$. □

K-core Example

We give an example of our method to compute the core number of nodes in a stream graph. The k-core is the maximal subgraph where all vertices have degree at least k. Interest in K-cores has risen through the years, they give a notion of density inside a graph and are "easily computable". These properties have numerous applications in the detection of patterns of interest, which can be very useful in anomaly detection [86], for instance.

In a stream graph, the k-core is defined as follows:

Definition 4.2.3. *The k-core of a stream graph $S = (T, V, W, E)$ is its largest cluster $C^k \subseteq W$ such that $\forall (t, v) \in C^k, d_t(v) \geq k$ in the induced sub-stream $S(C^k)$.*

The k-shell is a complementary notion defined as follows:

Definition 4.2.4. *The k-shell of a stream graph is the cluster $C^k \setminus C^{k+1}$ (the k-shell is defined only if the (k+1)-core is not empty : $C^{k+1} \neq \emptyset$).*

Now, we apply our framework to compute the time series corresponding to the core number of every node in the stream graph:

1. Compute the stable connected components.
2. For each connected component, a static graph is constructed, from the links in the stream graph induced by the corresponding component.
3. Batagelj's algorithm [11] is applied on the constructed static graphs which returns the core number of each node.
4. Finally, results for each component are concatenated to obtain the partition of W into k-cores.

Below, we provide the snippet of corresponding code in Straph. We use the Networkx implementation of Batagelj's algorithm:

Algorithm 1 Batagelj

Input : $G = (V, E)$, $degree$, $Neighbors$
Order nodes in V in increasing order according to their degree
for each $v \in V$ **do**
    $core[v] := degree[v]$
    for each $w \in Neighbors(v)$ **do**
        if $degree[w] > degree[v]$ **then**
            $degree[w] = degree[w] - 1$
            Reorder V accordingly
return *core*

k	Nb of node segments	% of $\|W\|$
1	967 165	99.4
2	10 377	0.6
3	36	0.00038

Table 4.3: Characteristics of the 3-core, 2-shell and 1-shell in the *Facebook* dataset. The total running time of the DAG parallel framework, with 8 cores, was 32.04s.

```
import networkx
S = stream_graph()
# Computation of the stable connected components of S:
stable_comp = S.stable_connected_components()
# We set the number of processes to 8 with 'n_jobs = 8'
kcores = S.graph_property(networkx.algorithms.core.core_number,
            stable_components=stable_comp,
            n_jobs=8)
```

We define the coreness of a stream graph node by:

$$c(v) = \sum_{C^k \subseteq W} \sum_{(I,X) \in C^k, v \in X} \frac{|I| \cdot k}{|T|}$$

Figure 4-12 shows the core number of temporal nodes, we can notice that 2-shell and 3-core tend to be distributed along horizontal and vertical lines. Horizontal lines indicate nodes frequently present in the 2 or 3-core and vertical lines represent specific periods of time. Figure 4-13 shows the distribution of the nodes corenesses in the *Facebook* dataset. In Table 4.3, we give the principal characteristics of the k-cores and k-shells.
(We do not provide an advanced analysis of core properties in stream graphs. This example aims to demonstrate the efficiency of this parallel framework, provided in Straph, in practice.)

This framework also allows us to easily compute the time series corresponding to a temporal node's core number (see Figure 4-14). The corresponding snippet of code in Straph is presented below:

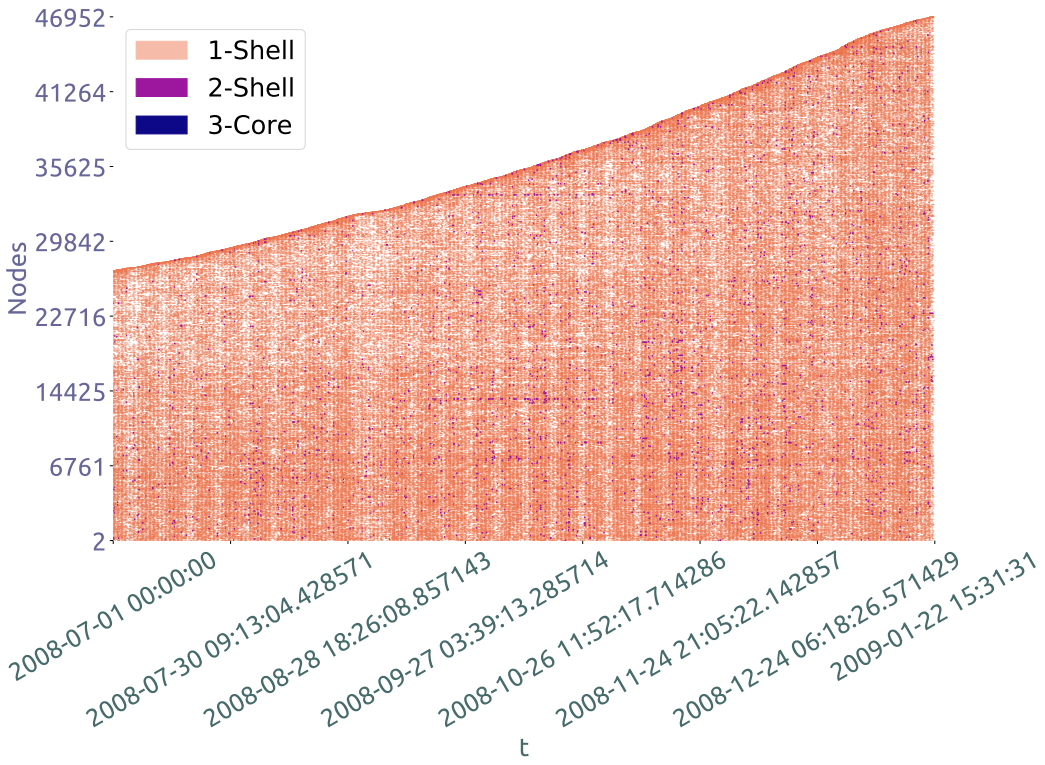

Figure 4-12: K-cores in a subset of the *Facebook* dataset (from the 01/07/2008 to the 22/01/2009).

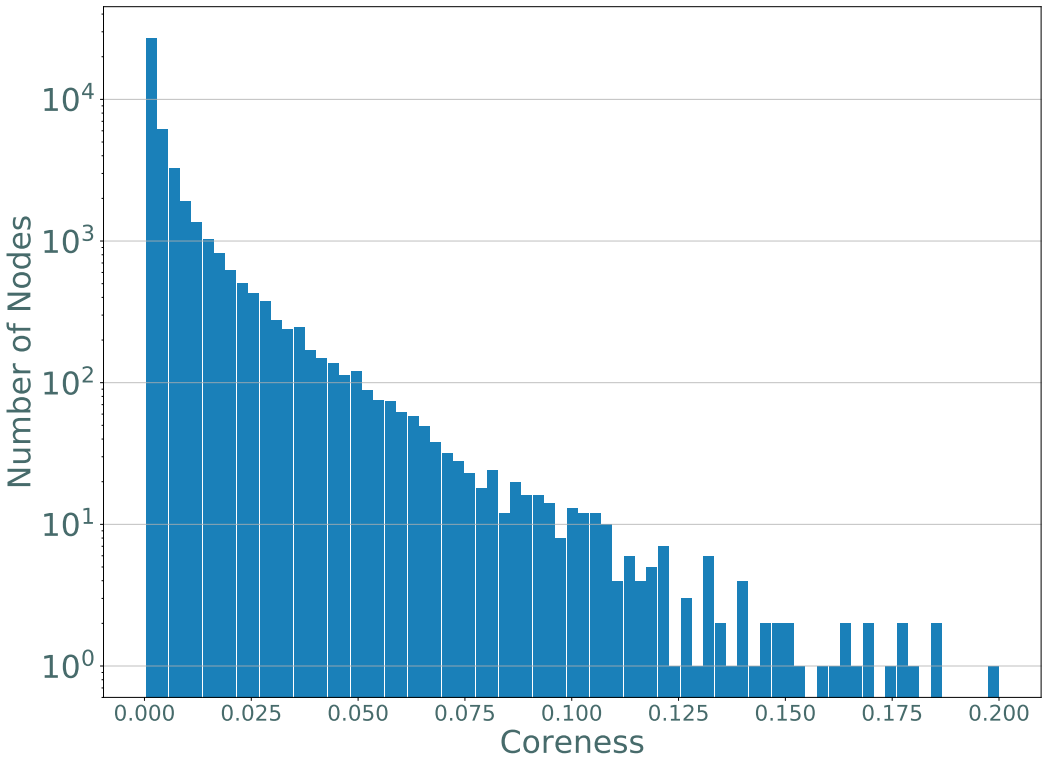

Figure 4-13: Coreness distribution in the facebook dataset

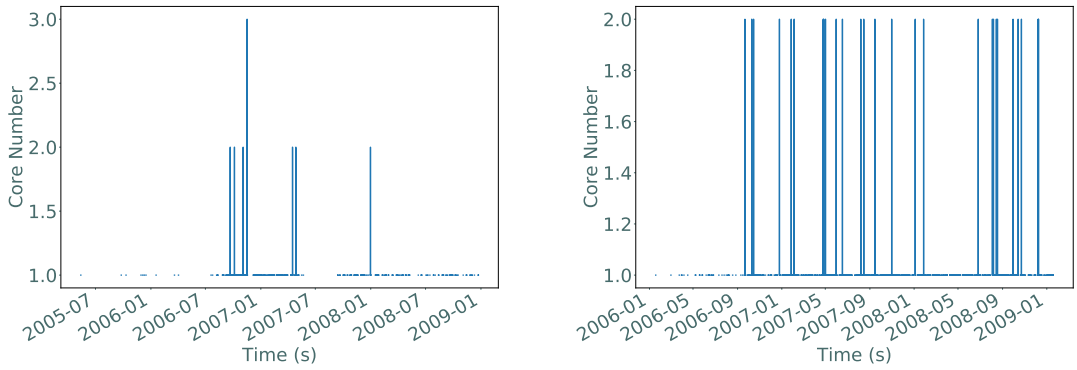

Figure 4-14: Core Number of nodes '1055' (left) and '2420' (right) over time in the *facebook* dataset.

```
import networkx
S = stream_graph()
# The following line of code returns a python dictionary associating
    each node to the time series of its core number over time.
node_to_signal = S.graph_property(nx.algorithms.core.core_number,
                  format="signal",postprocess=True)
```

With this framework we have shown that graph algorithms can be efficiently used to compute stream graph properties. Any stream graph property consistent regarding graph theory can be computed using this framework. The corresponding complexities are bounded by the size of the stream graph and the original complexities of the graph algorithm. Hence, this framework could be used as a baseline for any stream graph algorithm aiming to compute a graph theory consistent property.
Furthermore, the obtained complexities make it possible to compute properties that have not been computed before, up to our knowledge, on datasets with millions of nodes and links. The extracted time series could be used to better understand a particular dynamic or a given pattern. We hope to pursue this work in order to adapt this framework for anomaly detection in stream graphs.

4.3 Δ-Approximation

The main motivation of the approximation scheme, presented below, is to decrease the size of the condensation of a stream graph and facilitate its analysis. As we will demonstrate in the following, this approximation also has interesting properties, allowing many algorithms to perform better.

4.3.1 Approximate Strongly Connected Components

As explained in chapter 3, the fact that link segments start and end at slightly different times induces many strongly connected components of very low duration, that have little interest. We therefore propose to consider the following approximation of the stream graph $S = (T, V, W, E)$.

Let us first remind that δ is the minimal duration of a node or link segment used in the stream graph construction, see Section 1.3. Given an approximation parameter $\Delta < \delta$ and any time t in T, we define $\lfloor t \rfloor_\Delta$ as $\Delta \cdot \lfloor \frac{t}{\Delta} \rfloor$ and $\lceil t \rceil_\Delta$ as $\Delta \cdot \lceil \frac{t}{\Delta} \rceil$. We then define $S_\Delta = (T, V, W_\Delta, E_\Delta)$ where:

$$W_\Delta = \bigcup_{([b,e],v) \in \overline{W}} [\lceil b \rceil_\Delta, \lfloor e \rfloor_\Delta] \times \{v\}$$

and

$$E_\Delta = \bigcup_{([b,e],uv) \in \overline{E}} [\lceil b \rceil_\Delta, \lfloor e \rfloor_\Delta] \times \{uv\}$$

In other words, we replace each node segment $([b, e], v)$ by a shorter node segment that

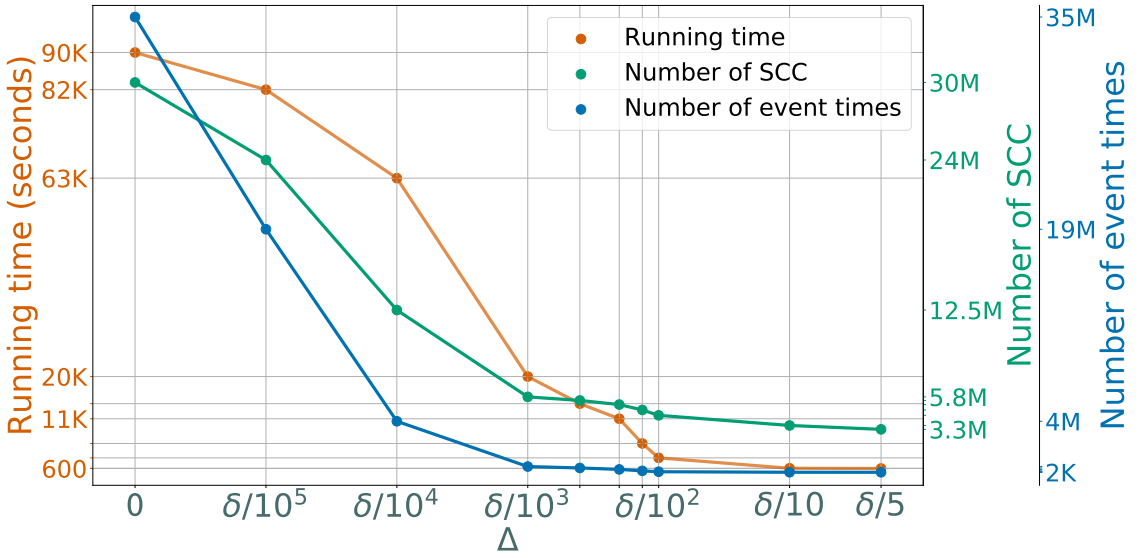

Figure 4-15: Running time of *SCC Direct*, number of SCC and number of event times in MawiLab, as a function of Δ (here, $\delta = 2s$).

starts at the first time after b and ends at the last time before e which are multiples of Δ. We proceed similarly with link segments.

First notice that S_Δ is an approximation of S, in the sense that S_Δ may be computed from S, but not the converse. In addition, each node or link segment in S lasts at least δ, and since Δ is lower than δ, no node or link segment disappears when S is transformed into S_Δ; only their starting and ending times change. Finally, S_Δ is included in S: $W_\Delta \subseteq W$ and $E_\Delta \subseteq E$. This has an important consequence: all paths in S_Δ are also paths in S, and so the approximation does not create any new reachability relation. It therefore preserves key information contained in the original stream, and we will show below that this information remains precise.

4.3.2 Experiments

Let us first observe the effect of the approximation, applied on the *Mawilab* dataset (see section 1.3), on strongly connected components in Figure 4-15. The number of components rapidly drops from its initial value of 30 millions (for $\Delta = 0$, *i.e.* no approximation) to less than 6 millions for $\Delta = \delta/10^3 = 0.002$. Its decrease is much slower when Δ grows further, which indicates that the stream does not anymore contain an important number of irrelevant components due to the *frontier effect*. As expected, this has a strong impact on computation time, which we also display; it also very rapidly drops, from more than one day to less than one hour, making computations on such large-scale datasets much easier.

Figure 4-16 presents the effect of Δ on size, duration and span distributions of strongly connected components. Initially, without approximation, the vast majority of com-

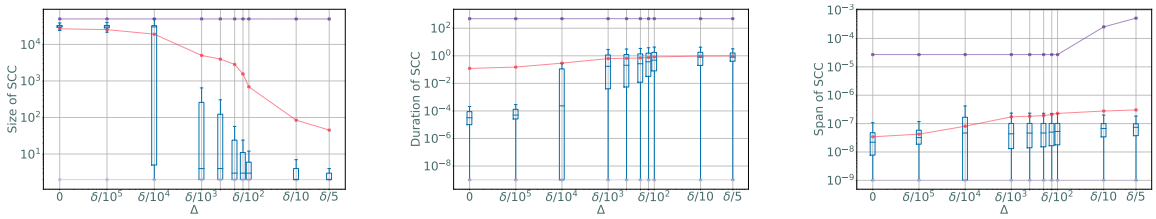

Figure 4-16: Box plots representing the distribution of the size (left), duration (middle) and span (right) of strongly connected components in Mawilab, for various values of Δ (here, $\delta = 2s$). We indicate the mean, minimal, and maximal values with dots connected by horizontal lines, as well as the median and percentiles with vertical boxes.

ponents involve tens of thousands nodes. For $\Delta = \delta/10^4$, we notice that while the number of components has decreased by half only fifty percent of them involve more than $30K$ nodes. Furthermore, as Δ increases, the number of components tends to be stable (Figure 4-15) but the number of components involving more than $30K$ nodes continues to drop. For $\Delta = \delta/10^3$, for instance, there are $5.8M$ SCC and among those, $760K$ involve more than $30K$ nodes. For $\Delta = \delta/10^2$, there are $4.4M$ SCC but among those, only $76K$ involve more than $30K$ nodes. This explains the differences observed in the execution time of *SCC Direct* (Figure 4-15) and confirms that the approximation eliminates most very short connected components, but not all: the ones which are not due to the frontier effect are preserved, another wanted feature.

4.3.3 Application to Latency Approximation

Although the approximation above has a strong impact on the number of strongly connected components, it preserves key information of the stream. We illustrate this by considering one of the most widely studied features of these objects: the latency between nodes [54, 112, 109, 47, 22]. Given two nodes u and v in a stream graph $S = (T, V, W, E)$, the latency from u to v is the minimal time needed to reach v from u by following links of S in a time-respecting manner, and taking into account node dynamics, see [59] for details.

Notice that latencies in S_Δ are necessarily larger than or equal to latencies in S, since paths in S_Δ are also paths in S. Therefore, latencies in S_Δ are upper bounds of latencies in S, and we show below that they are actually quite accurate approximations.

Figure 4-17 displays the average difference between latencies in S and S_Δ as a function of Δ for the Mawi dataset:

$$\frac{\sum_{u,v \in V, u \neq v} \ell_\Delta(u,v) - \ell(u,v)}{n \cdot (n-1)}$$

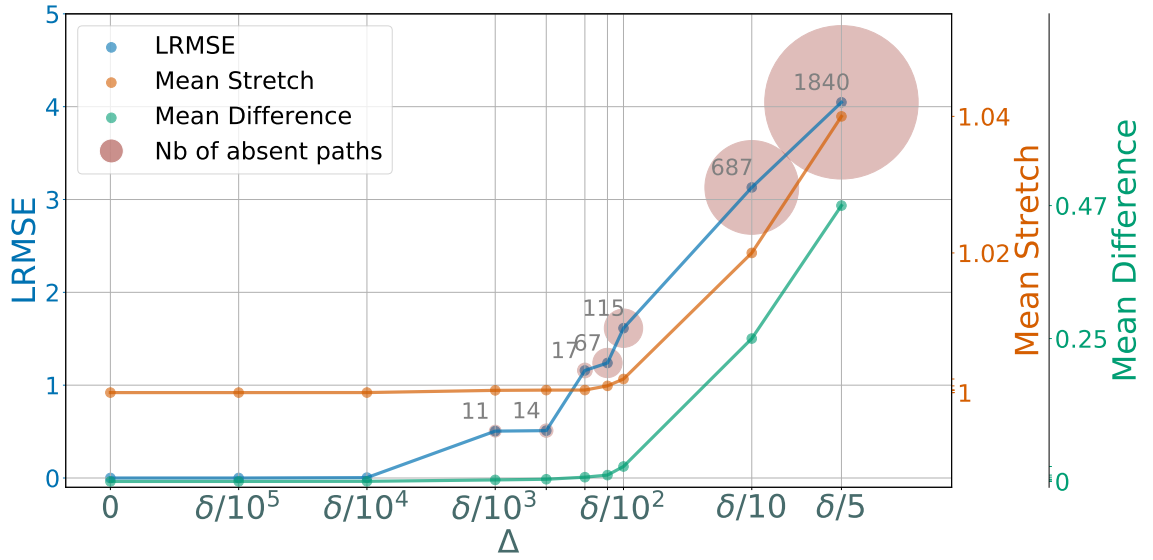

Figure 4-17: Evolution of the LRMSE, the average difference between latencies and the average latency stretch with respect to Δ in Mawilab. We indicate the number of missing paths and represent it as a disk of area proportional to this number.

It also displays the average latency stretch:

$$\frac{\sum_{u,v \in V, u \neq v} (\ell_\Delta(u,v)+1)/(\ell(u,v)+1)}{n \cdot (n-1)}$$

and the latency root mean square error ($LRMSE$):

$$LRMSE(S, S_\Delta) = \sqrt{\frac{\sum\limits_{u,v \in V, u \neq v} (\ell(u,v) - \ell_\Delta(u,v))^2}{n(n-1)}}$$

The figure also indicates the number of node pairs that were reachable in S but became unreachable in this approximation.

It clearly appears that latencies are not significantly impacted by approximation, thus confirming that S_Δ, despite its reduced number of strongly connected components, captures key information available in S. More precisely, only 11 temporal paths disappear for $\Delta = \delta/10^3$ and 115 disappear for $\Delta = \delta/10^2$, among a total number of $2,888,917$. The over-estimate of latencies is very small, with a $LRMSE$ of 0.51 and 1.61, respectively.

This has important consequences. For instance, one may compute latencies in S_Δ from its strongly connected components, which are much easier to compute and store than the ones of S, and obtain this way fast and accurate upper bounds (or approximations) of latencies in S, like we did here for the *Mawilab* dataset.

4.4 Discussion

We proposed three alternative representations for stream graphs, each one serving its own purpose. The condensation allows computing reachability in linear time. The stable DAG permits the design of a parallel framework to efficiently compute stream graph properties. Our approximation scheme, the Δ-approximation, enables faster computations while preserving key properties of the original data.

In the future we hope to provide other data representations with different applications, specifically toward community detection in stream graphs. As discussed, our representations also leave room for improvement. The condensation could be combined to a labelling scheme to further speed up queries. It could also be adapted to support different kinds of queries, by computing additional information and store it as condensation nodes labels for instance. The stable DAG could be represented differently, where a stable connected component would be defined relatively to the ones preceding it in the DAG. One may also push our dataset approximation scheme further, for instance by discarding node pairs with many (necessarily short) link segments between them, or nodes that are rarely present. Such approximations may lead to the emergence of strongly connected components with much larger span, and would open the way to coarse-grain description of stream graphs similar to the bow-tie model for the web [20].

5 Temporal Paths

Contributions

- The first single source one-pass polynomial time algorithm computing all kinds of optimal temporal paths in stream graphs: the *L-algorithm*
- Condensation based linear time algorithms for the computation of foremost and fastest paths

In this chapter, we focus on *paths*, a major concept in graph theory. Path algorithms and associated challenges are widely addressed in the scientific literature, see for instance [29, 26, 13] and [31]. Applications are numerous and of critical importance, such as computing the shortest path from an entity to another in a communication network [76], or in a transportation network [70].

Path notions in temporal or dynamic networks were proposed in [12, 47, 112, 109]. Temporal paths algorithms have been described in [109, 112, 78, 21, 96]. However, as mentioned in Chapter 1, the stream graph modelisation differs from existing temporal network formalisms. Consequently, different notions of stream graph paths have been proposed in [59] and arising algorithmic challenges have yet to be solved. Some path algorithms for link streams were proposed in [89] but their complexities were prohibitive. They cannot be applied on stream graphs with tens of millions of links.

In this chapter several strategies are evaluated in order to propose efficient solutions for temporal path problems in stream graphs. Many real world complex systems with a temporal dimension are conducive to a stream graph modelisation (see Chapter 1). Designing efficient stream graph algorithms for path computation is of major importance in view of the many concepts stemming from it. These concepts, such as betweenness centrality, among many others, have led to important advances in graph theory. Consequently, the work presented in this chapter could facilitate the analysis many real world temporal graphs.

Firstly, after defining stream graph paths (section 5.1), we present different types of optimal temporal path problems occurring in stream graphs (section 5.2). Then

we propose a single procedure, the *L-Algorithm* (section 5.3), to efficiently compute every kind of temporal path. We also introduce algorithms using the condensation of a stream graph, defined in chapter 4, to further reduce complexities of temporal paths computations (section 5.4). Finally, we evaluate our algorithms on 14 real world datasets (section 5.5).

5.1 Definitions

The following definitions of stream graph paths were introduced in [59].

Definition 5.1.1. *In a stream graph $S = (T, V, W, E)$ a **path** P from $(\alpha, u) \in W$ to $(\omega, v) \in W$ is a sequence (t_0, u_0, v_0), (t_1, u_1, v_1), ..., (t_k, u_k, v_k) of elements of $T \times V \times V$ such that $u_0 = u$, $v_k = v$, $t_0 \geq \alpha$, $t_k \leq \omega$, for all i, $t_i \leq t_{i+1}$, $v_i = u_{i+1}$ and $(t_i, u_i v_i) \in E$, $[\alpha, t_0] \times u \subseteq W$, $[t_k, \omega] \times v \subseteq W$, and for all i, $[t_i, t_{i+1}] \times v_i \subseteq W$. The term source refers to an element $u \in V$, a node which is the beginning of a path ($u = u_0$). A temporal source refers to an element $(t, u) \in W$, a temporal node which is the beginning of path ($[t, t_0] \times u \subseteq W$ and $u = u_0$). The start, refers to the effective beginning of a path i.e. the temporal node $(t, u) \in W$ such as $t = t_0$ and $u = u_0$; t is called the start time.*
Respectively, regarding the ending of a path, we use the terms: destination, temporal destination, arrival and arrival time.

For example, in Figure 5-1 (middle), $P = (4, A, B), (6, B, C), (7, C, E), (7, E, F)$ is a path from *source* A to *destination* F. The *start* is $(4, A)$ and the *arrival* is $(7, F)$; the *start time* and *arrival time* are equal to 4 and 7. In Figure 5-1 (top), $P = (3, A, B)$, $(6, B, C)$, $(7, C, E)$, $(7, E, F)$ is a path from the *temporal source* $(0, A)$ to the *temporal destination* $(7, F)$: the *start* is $(3, A)$ and the *arrival* is $(7, F)$.

In a given stream graph **there may exist an infinite number of temporal paths** from a *temporal source* to a *temporal destination*. For instance, in Figure 1-1, (t, A, B), $(4, B, E)$, $(7, E, F)$ is a valid path from $(0, A)$ to $(7, F)$ for any time $t \in [0, 4]$. These observations are also valid if we consider nodes as *source* or *destination* instead of *temporal source* or *temporal destination*.

5.2 Optimal Temporal Paths Problems

The addition of a temporal dimension leads to two paths properties, one relating to the **length** of a path, the number of steps in a path - as in graph theory - and another one relating to the **duration** of a path, the time it takes to travel the entire path. Given a temporal path P, we define three types of path characteristics: one for the **start time** from the *source*, denoted by $\mathcal{S}(P)$, another for the **arrival time** to the *destination*, $\mathcal{A}(P)$, and a third one for the **length** of the path, $\mathcal{D}(P)$. The **duration** of a temporal path is determined by $\mathcal{A}(P) - \mathcal{S}(P)$. For instance, given a path $P = (t_0, u, v_0)$, ..., (t_k, u_k, v) from (α, u) to (ω, v), we have: $\mathcal{A}(P) = t_k$,

$\mathcal{S}(P) = t_0$, its duration is equal to $t_k - t_0$ and its length to $\mathcal{D}(P) = k + 1$.

Given a *source*, u, and a *destination*, v, an **optimal temporal path problem** consists in finding paths that minimize a function $\mathcal{F}$ of $\mathcal{S}$, $\mathcal{A}$, $\mathcal{D}$ over the set of existing temporal paths from u to v, denoted by $\mathcal{P}_{uv}$. We call such a function a **path objective function**.

The number of optimal temporal paths between from a source (or a temporal source) to a destination (or a temporal destination) can be infinite. For instance, in Figure 1-1, (t, A, B), $(4, B, E)$ is a shortest path from $(0, A)$ to $(4, E)$ for any time $t \in [0, 4]$. Thus **we focus on the features of such paths and we do not try to enumerate them.**

The notion of temporal path raises several algorithmic questions. One question is to compute the quickest way to reach a *destination* (or a *temporal destination*) from a *temporal source*. This leads to the following definition:

Definition 5.2.1. *The **time to reach** (t, v) from u at time α, denoted by $\mathcal{T}_\alpha(u, (t, v))$, is defined as follows: $\mathcal{T}_\alpha(u, (t, v)) = \omega - \alpha$ where $\omega \leq t$ is the minimal value such that there is a path from (α, u) to (ω, v) in S and $[\omega, t] \subset T_v$. Such a path is called a **foremost path** from (α, u) to (t, v).*
Similarly, $T_\alpha(u, v)$ is the time to reach v from (α, u), without a time constraint on the destination node v; the associated path is a foremost path from (α, u) to v.
*A **foremost path problem** consists in finding a path that minimize $\mathcal{F}_{FoP} = \mathcal{A}$.*

For instance, on Figure 5-1 (top), $P = (3, A, B)$, $(6, B, C)$, $(7, C, E)$, $(7, E, F)$ is a foremost path from $(0, A)$ to F and the corresponding time to reach is $\mathcal{T}_0(A, F) = 7$.

One may also minimize the duration of a path:

Definition 5.2.2. *A path from (α, u) to (ω, v) is called a **fastest path** if it has minimal duration, and this duration is called the **latency** from (α, u) to (ω, v), denoted by $\ell((\alpha, u), (\omega, v))$. The latency $\ell(u, v)$ is the minimal latency for all $\alpha, \omega \in T$:*

$$\ell(u, v) = \min_{\alpha \in T, \omega \in T} (\ell(\alpha, u), (\omega, v))).$$

A path that has duration $\ell(u, v)$ is a fastest path from u to v.
*A **fastest path problem** consists in finding a path that minimize $\mathcal{F}_{FP} = \mathcal{A} - \mathcal{S}$.*

For instance, on Figure 5-1 (middle), $P = (4, A, B)$, $(6, B, C)$, $(7, C, E)$, $(7, E, F)$ is a fastest path from A to F and the corresponding latency is $\ell(A, F) = 3$.

As in a static graph, we may want to minimize the number of steps to reach a node from another:

Definition 5.2.3. *A path from (α, u) to (ω, v) is called a **shortest path** if it has minimal length, and this length is called the **distance** from (α, u) to (ω, v), denoted*

by $\delta((\alpha,u),(\omega,v))$. *The distance* $\delta(u,v)$ *is the minimal distance for all* $\alpha,\omega \in T$*:*

$$\delta(u,v) = \min_{\alpha\in T,\omega\in T}(\delta((\alpha,u),(\omega,v))).$$

A path that has length $\delta(u,v)$ *is a shortest path from* u *to* v*.*
A ***shortest path problem*** *consists in finding a path that minimize* $\mathcal{F}_{SP} = \mathcal{D}$*.*

For instance, on Figure 5-1 (bottom) $P = (4,A,B),(6,B,C),(9,C,D)$ is a shortest path from A to D and the corresponding distance is $\delta(A,D) = 3$.

In section 5.3 we will propose an algorithm solving the foremost path (FoP), shortest path (SP) and fastest path (FP) problems. We recall here the according objective functions:

$$\mathcal{F}_{FoP} = \mathcal{A}$$
$$\mathcal{F}_{SP} = \mathcal{D}$$
$$\mathcal{F}_{FP} = \mathcal{A} - \mathcal{S}$$

5.2.1 Multi-criteria optimal temporal paths

Combinations of these definitions make it possible to obtain **multi-criteria optimal temporal paths**: paths that minimize two criteria among length, duration and arrival time. We focus on **shortest foremost path** (foremost paths with minimal length, called SFoP), **fastest shortest paths** (shortest paths with minimal duration, called FSP) and **shortest fastest paths** (fastest paths with minimal length, called SFP).
The result of such functions can be a vector rather than a number. For instance, given any connected nodes $u,v \in V$ and a path $P \in \mathcal{P}_{uv}$, we can define $\mathcal{F}(P) = (\mathcal{S}(P),\mathcal{A}(P),\mathcal{D}(P))$. To compare two paths P and P' we compare the first component of $\mathcal{F}(P)$ and $\mathcal{F}(P')$; if their values are equal, we compare the values of the second component of $\mathcal{F}(P)$ and $\mathcal{F}(P')$ and so on.
The corresponding optimal temporal path problems consist in minimizing:

$$\mathcal{F}_{SFoP}(\cdot) = (\mathcal{A}(\cdot),\mathcal{D}(\cdot))$$
$$\mathcal{F}_{FSP}(\cdot) = (\mathcal{D}(\cdot),\mathcal{A}(\cdot) - \mathcal{S}(\cdot))$$
$$\mathcal{F}_{SFP}(\cdot) = (\mathcal{A}(\cdot) - \mathcal{S}(\cdot),\mathcal{D}(\cdot))$$

On Figure 5-2, $P = (2,A,B),(4,B,E),(7,E,F)$ (top) is a shortest foremost path from $(0,A)$ to F of length 3 and duration 7, $P = (4,A,B),(4,B,E),(7,E,F)$ (middle) is a shortest fastest path from A to F of length 3 and duration 3 and $P = (4,A,B),(4,B,E),(8,E,D)$ (bottom) is a fastest shortest path from A to D of length 3 and duration 4.

Remark 5.2.1. *Many other multi-criteria optimal temporal paths problems and their according objective functions can be defined. For instance, we can consider fastest*

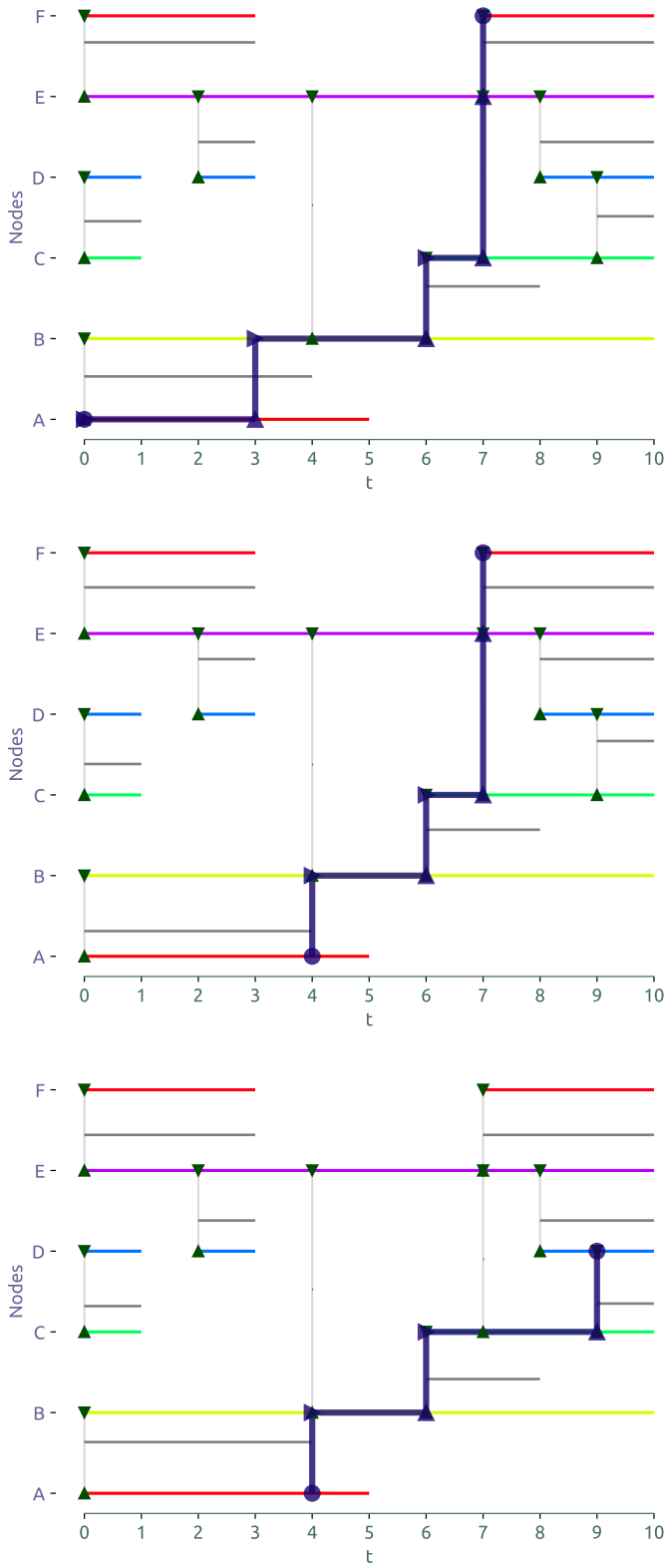

Figure 5-1: Examples of optimal temporal paths (top to bottom): *foremost path* from $(0, A)$ to F, *fastest path* from A to F, *shortest path* from A to D.

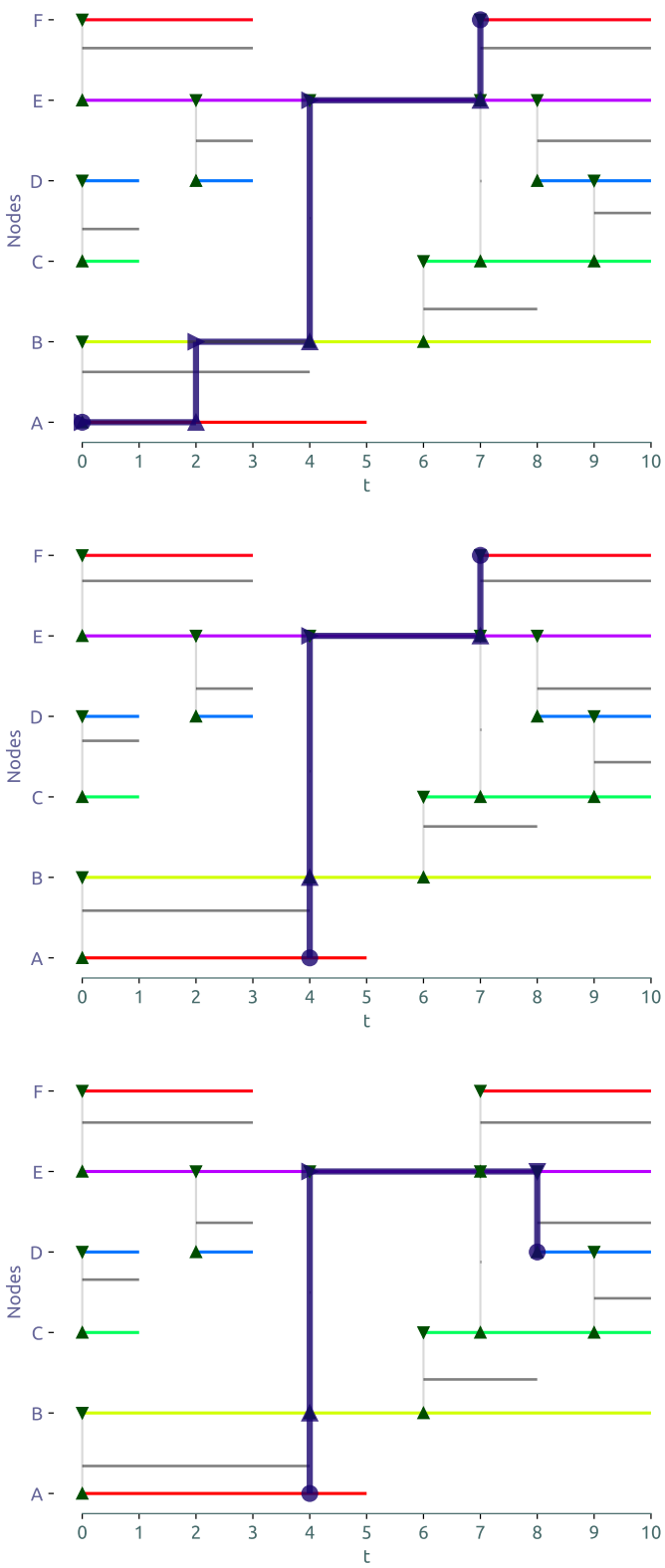

Figure 5-2: Examples of multi-criteria optimal temporal paths (top to bottom): *Shortest Foremost Path* $(0, A)$ to F, *Shortest Fastest Path* A to F, *Fastest Shortest Path* A to D.

foremost paths (foremost paths with minimal duration, called FFoP) and the according path problem which consist in minimizing $\mathcal{F}_{FFoP}(\cdot) = (\mathcal{A}(\cdot), \mathcal{A}(\cdot) - \mathcal{S}(\cdot))$. For the sake of clarity, int the following, we will focus on the paths problems defined previously.

5.2.2 Dominated Paths

The main difficulty in computing optimal temporal paths in stream graphs lies in the fact that, contrary to static graphs, **a subpath of an optimal temporal path may not be an optimal temporal path**.

For instance, in figure 5-1 (top) the subpath $P' = (3, A, B), (6, B, C), (7, C, E)$ of the foremost path $P = (3, A, B), (6, B, C), (7, C, E), (7, E, F)$ is not a foremost path from $(0, A)$ to E. Indeed E can be reached from $(0, A)$ at instant 4 with the following foremost path: $(3, A, B), (4, B, E)$.

As a result we cannot use the same strategy as in Dijkstra or Floyd-Warshall algorithms: **we cannot extend optimal temporal paths to obtain optimal temporal paths**. Therefore we focus on temporal paths that can be extended to obtain optimal temporal paths - whether or not they are optimal temporal paths themselves.

Definition 5.2.4. *Given two temporal paths $P, P' \in \mathcal{P}_{uv}$, P and P are **swappable** if and only if: $\exists w \in V \setminus \{v\}$, $\exists R \in \mathcal{P}_{vw}$ s.t $P \oplus R \in \mathcal{P}_{uw}$ and $P' \oplus R \in \mathcal{P}_{uw}$.*
Where $\oplus$ is the path concatenation operator.

If two paths P and P' are swappable, it means that both these paths can be extended by the same path. There exists at least one path O such that P and P' are both prefixes of O.

Remark 5.2.2. *Given two paths P, $P' \in \mathcal{P}_{uv}$ if $[min(\mathcal{A}(P), \mathcal{A}(P')), max(\mathcal{A}(P), \mathcal{A}(P'))] \not\subseteq T_v$ then P and P' cannot be swappable.*

In the stream graph S of Figure 5-1, let us consider $P' = (2, D, E)$, $P = (8, D, E)$ and $R = (9, E, F)$, P and P' are swappable. Indeed, $P \oplus R = (2, D, E), (9, E, F)$ and $P' \oplus R = (8, D, E), (9, E, F)$ both exist in S. On the other hand, $P = (2, E, F)$ and $P' = (7, E, F)$ are not swappable, as $[2, 7] \not\subseteq T_F$.

Definition 5.2.5. *Given $\mathcal{F}$, a path objective function, and two temporal paths $P, P' \in \mathcal{P}_{uv}$, P **$\mathcal{F}$-dominates** P' if and only if:*
P and P' are swappable and $\forall w \in V \setminus \{v\}$, $\forall R \in \mathcal{P}_{vw}$ s.t $P \oplus R \in \mathcal{P}_{uw}$ and $P' \oplus R \in \mathcal{P}_{uw}$, we have $\mathcal{F}(P \oplus R) \leq \mathcal{F}(P' \oplus R)$.

For instance, in Figure 5-1, the path from A to E for the objective function $\mathcal{F}_{FP} = \mathcal{A} - \mathcal{S}$ for fastest paths: $P' = (2, A, B), (4, B, E)$ is $\mathcal{F}_{FP}$-dominated by the path $P = (4, A, B), (4, B, E)$. Indeed, if we consider any node $w \in V$ and any path $R \in \mathcal{P}_{Ew}$ s.t $\mathcal{S}(R) \geq 4$, we would have $\mathcal{F}_{FP}(P \oplus R) < \mathcal{F}_{FP}(P' \oplus R)$ as $\mathcal{S}(P) > \mathcal{S}(P')$.

In other words, a dominated path may not be extended to obtain an optimal temporal path because it can be replaced (*swapped*) by another path resulting in a better, or equivalent, solution regarding the minimisation of $\mathcal{F}$.

Nevertheless **a dominated path can also be an optimal temporal path**. In Figure 5-1, $P' = (7, B, C), (7, C, E)$ is a fastest path from B to E and P' is $\mathcal{F}_{FP}$-dominated by the path $P = (8, B, C), (9, C, D), (9, C, D)$. In addition, we can notice that P is not a fastest path, an optimal temporal path can be dominated by a non-optimal temporal path.

Definition 5.2.6. *Given $\mathcal{F}$ a path objective function, $\mathcal{Q}$ is a **domination function** if and only if: $\forall u, v \in V$, $\forall P, P' \in \mathcal{P}_{uv}$ s.t P and P' are swappable and $\mathcal{Q}(P) \leq \mathcal{Q}(P') \iff P$ $\mathcal{F}$-dominates P'.*

For each type of optimal temporal paths problem, we will define a domination function $\mathcal{Q}$ of $\mathcal{S}, \mathcal{A}, \mathcal{D}$.

Lemma 5.2.1. *(**Fastest Path**) $\mathcal{Q}_{FP} = -\mathcal{S}$ is a domination function for the fastest path problem.*

Proof. $\implies$ Suppose that P and $P' \in \mathcal{P}_{uv}$ are swappable, that $\mathcal{Q}_{FP}(P) \leq \mathcal{Q}_{FP}(P')$ and that there exists $w \in V \setminus \{v\}$ and $R \in \mathcal{P}_{vw}$, such that:

$$\mathcal{F}_{FP}(P \oplus R) > \mathcal{F}_{FP}(P' \oplus R)$$

$$\mathcal{A}(P \oplus R) - \mathcal{S}(P \oplus R) > \mathcal{A}(P' \oplus R) - \mathcal{S}(P' \oplus R)$$

As $\mathcal{S}(P) \geq \mathcal{S}(P')$, we have $\mathcal{S}(P \oplus R) \geq \mathcal{S}(P' \oplus R)$, then: $\mathcal{F}_{FP}(P \oplus R) \leq \mathcal{F}_{FP}(P' \oplus R)$ which is a contradiction.

$\impliedby$ Suppose that P $\mathcal{F}_{FP}$-dominates P' since P and P' are swappable, there exists $w \in V \setminus \{v\}$ and at least a path $R \in \mathcal{P}_{uw}$ such that $\mathcal{F}_{FP}(P \oplus R) \leq \mathcal{F}_{FP}(P' \oplus R)$. As $\mathcal{A}(P \oplus R) = \mathcal{A}(P' \oplus R)$ we must have $\mathcal{S}(P) \geq \mathcal{S}(P')$ which implies $\mathcal{Q}_{FP}(P) \leq \mathcal{Q}_{FP}(P')$. □

Lemma 5.2.2. *(**Shortest Path**) A domination function for the shortest path problem is $\mathcal{Q}_{SP} = \mathcal{D}$.*

Proof. $\implies$ Suppose that P and $P' \in \mathcal{P}_{uv}$ are swappable, that $\mathcal{Q}_{SP}(P) \leq \mathcal{Q}_{SP}(P')$ and that there exists $w \in V \setminus \{v\}$ and $R \in \mathcal{P}_{vw}$ such that:

$$\mathcal{F}_{SP}(P \oplus R) > \mathcal{F}_{SP}(P' \oplus R)$$

$$\mathcal{D}(P \oplus R) > \mathcal{D}(P' \oplus R)$$

As $\mathcal{D}(P) \leq \mathcal{D}(P')$, we have $\mathcal{D}(P \oplus R) \leq \mathcal{D}(P' \oplus R)$, then: $\mathcal{F}_{SP}(P \oplus R) \leq \mathcal{F}_{SP}(P' \oplus R)$ which is a contradiction.

$\impliedby$ Suppose that P $\mathcal{F}_{SP}$-dominates P' since P and P' are swappable, there exists $w \in V \setminus \{v\}$ and at least a path $R \in \mathcal{P}_{uw}$ such that $\mathcal{F}_{SP}(P \oplus R) \leq \mathcal{F}_{SP}(P' \oplus R)$, i.e. $\mathcal{D}(P \oplus R) \leq \mathcal{D}(P' \oplus R)$. Therefore we must have $\mathcal{D}(P) \leq \mathcal{D}(P')$, i.e. $\mathcal{Q}_{SP}(P) \leq \mathcal{Q}_{SP}(P')$. □

Lemma 5.2.3. *(**Shortest Foremost Path**) $\mathcal{Q}_{SFoP} = \mathcal{D}$ is a domination function for the shortest foremost path problem.*

Proof. $\Longrightarrow$ Suppose that P and $P' \in \mathcal{P}_{uv}$ are swappable, that $\mathcal{Q}_{SFoP}(P) \leq \mathcal{Q}_{SFoP}(P')$ and that there exists $w \in V \setminus \{v\}$ and $R \in \mathcal{P}_{vw}$ such that:

$$\mathcal{F}_{SFoP}(P \oplus R) > \mathcal{F}_{SFoP}(P' \oplus R)$$

We have $\mathcal{A}(P \oplus R) = \mathcal{A}(P' \oplus R)$ and $\mathcal{D}(P \oplus R) \leq \mathcal{D}(P' \oplus R)$ as $\mathcal{D}(P) \leq \mathcal{D}(P')$, then: $\mathcal{F}_{SFoP}(P \oplus R) \leq \mathcal{F}_{SFoP}(P' \oplus R)$ which is a contradiction.

$\Longleftarrow$ Suppose that P $\mathcal{F}_{SFoP}$-dominates P' since P and P' are swappable, there exists $w \in V \setminus \{v\}$ and at least a path $R \in \mathcal{P}_{uw}$ such that $\mathcal{F}_{SFoP}(P \oplus R) \leq \mathcal{F}_{SFoP}(P' \oplus R)$. As $\mathcal{A}(P \oplus R) = \mathcal{A}(P' \oplus R)$ and $\mathcal{D}(P \oplus R) \leq \mathcal{D}(P' \oplus R)$ we must have $\mathcal{D}(P) \leq \mathcal{D}(P')$ which implies $\mathcal{Q}_{SFoP}(P) \leq \mathcal{Q}_{SFoP}(P')$. □

Lemma 5.2.4. *(**Fastest Shortest Path**) $\mathcal{Q}_{FSP}(\cdot) = (\mathcal{D}(\cdot), -\mathcal{S}(\cdot))$ is a domination function for the fastest shortest path problem.*

Proof. $\Longrightarrow$ Suppose that P and $P' \in \mathcal{P}_{uv}$ are swappable, that $\mathcal{Q}_{FSP}(P) \leq \mathcal{Q}_{FSP}(P')$ and that there exists $w \in V \setminus \{v\}$ and $R \in \mathcal{P}_{vw}$ such that:

$$\mathcal{F}_{FSP}(P \oplus R) > \mathcal{F}_{FSP}(P' \oplus R)$$

If $\mathcal{D}(P) < \mathcal{D}(P')$, we have $\mathcal{D}(P \oplus R) < \mathcal{D}(P' \oplus R)$, then: $\mathcal{F}_{FSP}(P \oplus R) < \mathcal{F}_{FSP}(P' \oplus R)$. If $\mathcal{D}(P) = \mathcal{D}(P')$ and $\mathcal{S}(P) \geq \mathcal{S}(P')$, we have $\mathcal{D}(P \oplus R) = \mathcal{D}(P' \oplus R)$ and $\mathcal{S}(P \oplus R) \geq \mathcal{S}(P' \oplus R)$, then: $\mathcal{F}_{FSP}(P \oplus R) \leq \mathcal{F}_{FSP}(P' \oplus R)$. In both cases we have a contradiction.

$\Longleftarrow$ Suppose that P $\mathcal{F}_{FSP}$-dominates P' since P and P' are swappable, there exists $w \in V \setminus \{v\}$ and at least a path $R \in \mathcal{P}_{vw}$ such that $\mathcal{F}_{FSP}(P \oplus R) \leq \mathcal{F}_{FSP}(P' \oplus R)$. If $\mathcal{D}(P \oplus R) < \mathcal{D}(P' \oplus R)$ then we have $\mathcal{D}(P) < \mathcal{D}(P')$. If $\mathcal{D}(P \oplus R) = \mathcal{D}(P' \oplus R)$ and $\mathcal{S}(P \oplus R) \geq \mathcal{S}(P' \oplus R)$ then we have $\mathcal{S}(P) \geq \mathcal{S}(P')$. Both cases implies $\mathcal{Q}_{FSP}(P) \leq \mathcal{Q}_{FSP}(P')$. □

Lemma 5.2.5. *(**Shortest Fastest Path**) $\mathcal{Q}_{SFP}(\cdot) = (-\mathcal{S}(\cdot), \mathcal{D}(\cdot))$ is a domination function for the shortest fastest path problem.*

Proof. $\Longrightarrow$ Suppose that P and $P' \in \mathcal{P}_{uv}$ are swappable, that $\mathcal{Q}_{SFP}(P) \leq \mathcal{Q}_{SFP}(P')$ and that there exists $w \in V \setminus \{v\}$ and $R \in \mathcal{P}_{vw}$ such that:

$$\mathcal{F}_{SFP}(P \oplus R) > \mathcal{F}_{SFP}(P' \oplus R)$$

If $\mathcal{S}(P) > \mathcal{S}(P')$, we have $\mathcal{S}(P \oplus R) > \mathcal{S}(P' \oplus R)$, then: $\mathcal{F}_{SFP}(P \oplus R) < \mathcal{F}_{SFP}(P' \oplus R)$. If $\mathcal{S}(P) = \mathcal{S}(P')$ and $\mathcal{D}(P) \leq \mathcal{D}(P')$ we have $\mathcal{S}(P \oplus R) = \mathcal{S}(P' \oplus R)$ and $\mathcal{D}(P \oplus R) \leq \mathcal{S}(P' \oplus R)$ then: $\mathcal{F}_{SFP}(P \oplus R) < \mathcal{F}_{SFP}(P' \oplus R)$. In both cases we have a contradiction.

$\Longleftarrow$ Suppose that P $\mathcal{F}_{SFP}$-dominates P', since P and P' are swappable, there exists $w \in V \setminus \{v\}$ and at least a path $R \in \mathcal{P}_{uw}$ such that $\mathcal{F}_{SFP}(P \oplus R) \leq \mathcal{F}_{SFP}(P' \oplus R)$. If $\mathcal{S}(P \oplus R) > \mathcal{S}(P' \oplus R)$ then we have $\mathcal{S}(P) > \mathcal{S}(P')$. If $\mathcal{S}(P \oplus R) = \mathcal{S}(P' \oplus R)$ and $\mathcal{D}(P \oplus R) \leq \mathcal{D}(P' \oplus R)$ then we have $\mathcal{D}(P) \leq \mathcal{D}(P')$. Both cases implies $\mathcal{Q}_{SFP}(P) \leq \mathcal{Q}_{SFP}(P')$. □

Remark 5.2.3. *A domination function for the foremost path problem assigns the same value (priority) to all paths. Suppose that P and $P' \in \mathcal{P}_{uv}$ are swappable and that there exists $w \in V \setminus \{v\}$ and $R \in \mathcal{P}_{vw}$ then, as $\mathcal{A}(P \oplus R) = \mathcal{A}(P' \oplus R)$ we always have $\mathcal{F}_{FoP}(P \oplus R) = \mathcal{F}_{FoP}(P' \oplus R)$. As long as two paths are swappable, one can be replaced by the other without impacting the value of the objective function, $\mathcal{F}_{FoP}$, for any extended path. Thus, $\mathcal{Q}_{Fop}(\cdot) = 0$ is a valid domination function for the foremost path problem.*

Remark 5.2.4. *In definition 5.2.5 it is necessary to have $\mathcal{F}(P \oplus R) \leq \mathcal{F}(P' \oplus R)$. Indeed, if we had $\mathcal{F}(P \oplus R) < \mathcal{F}(P' \oplus R)$, considering $\mathcal{F}_{FoP} = \mathcal{A}$, as $\forall R \in \mathcal{P}_{vw}$, $\mathcal{A}(P \oplus R) = \mathcal{A}(P' \oplus R)$, no path would dominates another.*

Consequently, in definition 5.2.6, it is necessary to have $\mathcal{Q}(P) \leq \mathcal{Q}(P')$. Indeed, considering $\mathcal{Q}_{FoP} = 0$ and two paths P and P' such that $\mathcal{A}(P) = \mathcal{A}(P')$, we have: $P\mathcal{F}_{FoP}$-dominates $P' \iff \mathcal{Q}_{FoP}(P) = \mathcal{Q}_{Fop}(P')$.

5.3 L-Algorithm

In the following we propose a new algorithm, the **L-Algorithm**, inspired by the algorithms introduced in [109], covering continuous interactions as well as nodes dynamics, in order to solve stream graph optimum temporal path problems.

As mentioned, the number of optimal temporal paths from a node (or a temporal node) to a node (or a temporal node) can be infinite: we solely focus on their properties. A path P from u (or (t, u)) to v (or (t, v)) can be represented by a triplet (s, a, d) where $s = \mathcal{S}(P)$ is the starting time from u (or (t, u)), $a = \mathcal{A}(P)$ is the arrival time in v (or (t, v)) and $d = \mathcal{D}(P)$ the corresponding length. Path objective functions, $\mathcal{F}$, and domination functions, $\mathcal{Q}$ are functions of $\mathcal{S}$, $\mathcal{A}$ and $\mathcal{D}$: we can apply them, without a loss of generality, on triplets (s, a, d).

Given a set of temporal links (b_0, e_0, u_0, v_0), ..., (b_n, e_n, u_n, v_n), we represent the temporal paths $\{(t_0, u_0, v_0)$, ..., $(t_n, u_n, v_n)\}$ with $t_0 \in [b_0, e_0]$, ..., $t_n \in [b_n, e_n]$ that go through these links by a canonical triplet: the one with the maximal starting time and minimal arrival time:

$$(\min_{0 \leq j \leq n} e_j, \max_{0 \leq j \leq n} b_j, n + 1)$$

For instance, in the stream graph of Figure 5-1, the set of temporal paths going through the temporal links $(0, 4, A, B)$, $(6, 8, B, C)$, $(9, 10, C, D)$ corresponds to the canonical triplet $(4, 9, 3)$. Such a triplet may correspond to an absurd path P, with $\mathcal{S}(P) > \mathcal{A}(P)$, but only triplets corresponding to existing paths will be outputted, as

we will show in the following. Notice, moreover, that all paths that optimize one of our domination functions correspond to such a triplet.

Our *L-Algorithm* computes the path optimal with regard to a given objective function - and a corresponding domination function - from a single source (t, x). For all $u \in V$, the following data structures are maintained:

- L_u: a triplet (s_u, a_u, d_u) representing a temporal path, that may be extended to form an optimal temporal path, from the *source* to u. s_u correspond to the *starting time* from the *source*, a_u to the *arrival time* in u and d_u to the distance from the *source* to u; this is the path that minimizes the domination function among all paths arriving at u no later than the current event time, on u's current node segment.
- A_u: a temporal adjacency list, containing the current neighbors of u as well as the end time of their interactions; A is updated as links appear and disappear.
- R_u: a triplet (s_u, a_u, d_u), currently minimizing $\mathcal{F}$, the path objective function.

For each event time c, better paths than those previously observed may be caused either by the appearance of a new link, or by the fact that they start from x at time c. Therefore, we use a procedure inspired by Dijkstra's algorithm. This algorithm maintains a set of links (e, w, y) such that: w is an observed node, y is one of its neighbours at the current event time, and e is the latest possible starting time for a path from (t, x) to (c, w). Each path is associated with a priority which is the value of the domination function for the path built by adding the link (c, w, y) to the best observed path from (t, x) to (c, w). When the algorithm terminates we return R_u for all $u \in V$.

Example

As an example we apply the *L-Algorithm* to the shortest path problem, with the objective function $\mathcal{F}_{SP} = \mathcal{D}$ and the domination function $\mathcal{Q}_{SP} = \mathcal{D}$. We compute the distances from the temporal source $(0, A)$ to every other node in the stream graph of Figure 5-1. We will denote by $(+, t_0, t_1, X, Y)$ the arrival of a link from t_0 to t_1 between nodes X and Y. The values stored in A, I, L and R at each event time are detailed in Table 5.1. We do not show the steps regarding nodes and links departures (lines 24-27) as they do not raise any particular difficulties.

- L_A is initialised with the canonical triplet $(5, 0, 0)$ and R_A with the triplet $(0, 0, 0)$ (lines 5 and 8).
- $(+, 0, 1, C, D)$, $(+, 0, 3, E, F)$, $(+, 0, 4, A, B)$: the temporal adjacency list is updated, $(1, C)$ is added to A_D, $(1, D)$ to A_C, $(3, E)$ to A_F, $(3, F)$ to A_E and $(4, B)$ is added to A_A and $(4, A)$ to A_B (line 13). As C, D, E and F aren't connected with A at time 0: L_C, L_D, L_E and L_F remain empty (lines 14, 17).
  L_A is not empty: $(4, A, B)$ is added to I with priority 1 ($\mathcal{Q}_{SP}(L_A)+1 = (d_A+1 = 1)$ (lines 14-19). During Algorithm 3 the node A is marked as visited (line 6).

Algorithm 2 Given a stream graph $S = (T, V, W, E)$, return features (length, starting and arrival times) of optimal temporal paths, according to a path objective function $\mathcal{F}$ and a domination function $\mathcal{Q}$, from a temporal source node $(t, x) \in W$ to all nodes in V.

Def: $L\_Algorithm(S, (t, x), \mathcal{F}, \mathcal{Q})$
Input: $(t, x) \in W$ a temporal source, E^* the ordered set of event times, $\mathcal{F}$ an objective function and $\mathcal{Q}$ a domination function, both corresponding to the type of optimal temporal path.
Output: Features (starting time, arrival time, length) of optimal temporal paths from (t, x) to all nodes in V.
if $\exists [\beta_x, \theta_x] \in \overline{T_x}$ s.t. $t \in [\beta_x, \theta_x]$ **then**
    $L_x = [(\theta_x, t, 0)]$
else return ▷ No path can start at (t, x)
$L_v \leftarrow \emptyset$ for all $v \in V \setminus \{x\}$
$R_x = (t, t, 0)$ and $R_v \leftarrow (inf, inf, inf)$ for all $v \in V \setminus \{x\}$
Initialize A_v to $\emptyset$ for all $v \in V$
for c **in** Π **do**
    $I \leftarrow \emptyset$ ▷ Priority Queue
    for all link arrivals (c, e, u, v) **do**
        Insert (e, v) in A_u and (e, u) in A_v
        if $L_u \neq \emptyset$ **then** ▷ u connected with the source
            $(s_u, a_u, d_u) = L_u$
            Add (e, u, v) to I with priority $\mathcal{Q}((min(s_u, c), c, d_u + 1))$.
        if $L_v \neq \emptyset$ **then** ▷ v connected with the source
            $(s_v, a_v, d_v) = L_v$
            Add (e, v, u) to I with priority $\mathcal{Q}((min(s_v, c), c, d_v + 1))$.
    if $A_x \neq \emptyset$ **then**
        For all (τ, w) in A_x add (τ, x, w) to I with priority $\mathcal{Q}((min(s_x, e), c, d_x + 1))$
    if $I \neq \emptyset$ **then**
        $L = Dijkstra\_Update(c, L, I, A, \mathcal{F}, \mathcal{Q})$
    for all link departures (c, u, v) **do**
        Remove (e, v) from A_u and (e, u) from A_v
    for all node departures (b, c, u) **do**
        $L_u \leftarrow \emptyset$
return R_v for all $v \in V$

Events	**A**	**I**	**L**	**R**
Initialisation			$L_A = (5, 0, 0)$	$R_A = (0, 0, 0)$
$(+, 0, 1, C, D)$ $(+, 0, 3, E, F)$ $(+, 0, 4, A, B)$	$A_D = [(1, C)]$ $A_C = [(1, D)]$ $A_F = [(3, E)]$ $A_E = [(3, F)]$ $A_A = [(4, B)]$ $A_B = [(4, A)]$	$[(1, (4, A, B))]$	$L_B = (4, 0, 1)$	$R_B = (0, 0, 1)$
$(+, 2, 3, D, E)$	$A_D = [(3, E)]$ $A_E = [(3, F),$ $(3, D)]$			
$(+, 4, 4, B, E)$	$A_B = [(4, A),$ $(4, E)]$ $A_E =$ $[(4, B)]$	$[(2, (4, B, E))]$	$L_E = (4, 4, 2)$	$R_E = (4, 4, 2)$
$(+, 6, 8, B, C)$	$A_B = [(8, C)]$ $A_C = [(8, B)]$	$[(2, (8, B, C))]$	$L_C = (4, 6, 2)$	$R_C = (4, 6, 2)$
$(+, 7, 10, E, F)$ $(+, 7, 10, D, E)$ $(+, 7, 8, A, B)$	$A_E =$ $[(10, D),$ $(10, F)]$ $A_F = [(10, E)]$ $A_D =$ $[(10, E)]$ $A_A = [(8, B)]$ $A_B = [(8, A),$ $(8, C)]$	$[(2, (8, B, A)),$ $(3, (10, E, F)),$ $(3, (10, E, D))]$	$L_A = (4, 7, 2)$ $L_F = (4, 7, 3)$ $L_D = (4, 7, 3)$	$R_F = (4, 7, 3)$ $R_D = (4, 7, 3)$
$(+, 8, 10, D, F)$	$A_D =$ $[(10, E),$ $(10, F)]$ $A_F = [(10, E),$ $(10, D)]$	$[(2, (10, E, D)),$ $(2, (10, E, F))]$		
Termination				Return R

Table 5.1: Example of *L-Algorithm* for shortest paths ($\mathcal{F}_{SP}$ and $\mathcal{Q}_{SP}$).

Algorithm 3 Update L for all nodes in A

1: **Def:** $Dijkstra\_Update(c, L, I, A, \mathcal{F}, \mathcal{Q})$
2: **Input:** c is the current event time, t is the starting time from the source, L the structure containing current paths, A the temporal adjacency list of the stream graph, I a priority queue, $\mathcal{F}$ a path objective function and $\mathcal{Q}$ a domination function.
3: **while** I **is not empty do**
4: $(e, w, y) = I.pop()$
5: $(s_w, a_w, d_w) = L_w$
6: Mark w as *visited*
7: s_y, a_y, $d_y = min(e, s_w)$, c, $d_w + 1$
8: **if** $\mathcal{F}((min(s_y, a_y), a_y, d_y)) < \mathcal{F}(R_y)$ **then**
9: $R_y = (min(s_y, a_y), a_y, d_y)$
10: **if** $L_y \neq \emptyset$ **then**
11: $(s'_y, a'_y, d'_y) = L_y$
12: **if** $L_y = \emptyset$ **or** $\mathcal{Q}((min(s_y, c), a_y, d_y)) < \mathcal{Q}((min(s'_y, c), a'_y, d'_y))$ **then**
13: $L_y = (s_y, a_y, d_y)$
14: **for** (e, h) **in** A_y **do**
15: **if** h **is not marked as** *visited* **then**
16: Add (e, y, h) to I with priority $\mathcal{Q}((min(s_y, c), c, d_y + 1))$
return L

L_B is updated with $s_B = min(4, s_A) = 4$, $a_B = min(0, 0) = 0$, $d_B = d_A + 1 = 1$ (line 7 and 12-13). $(min(s_B, a_B), a_B, d_B) = (0, 0, 1)$ is added to R_B (line 8-9). As A is already marked as visited, $(4, B, A)$ is not added to I (lines 14-15). I is empty, proceed to the next link.

- $(+, 2, 3, D, E)$: A_D and A_E are updated. L_D and L_E are empty, proceed to the next link.
- $(+, 4, 4, B, E)$: A_B and A_E are updated. L_B is not empty, $(4, B, E)$ is added to I with priority 2 ($\mathcal{Q}_{SP}(L_B) + 1 = (d_b + 1 = 2)$. L_E is updated with $s_E = min(4, s_B) = 4$, $a_E = 4$, $d_E = 2$ and $R_E = (4, 4, 2)$. $(4, B, A)$ is added to I. L_A and R_A are not updated, as $\mathcal{Q}_{SP}((4, 0, 0)) < \mathcal{Q}_{SP}((4, 4, 3))$ and $\mathcal{F}_{SP}((4, 0, 0)) < \mathcal{F}_{SP}((4, 4, 3))$.
- $(+, 6, 8, B, C)$: A_B and A_C are updated. $L_B \neq \emptyset$, $(8, B, C)$ is added to I and B is marked as visited. L_C is updated with $s_C = min(8, s_B) = 4$, $a_C = 6$, $d_C = 2$ and $R_C = (4, 6, 2)$. B is already marked as visited, I is empty.
- $(+, 7, 10, E, F)$, $(7, 10, D, E)$, $(7, 8, A, B)$: A_E and A_F are updated. $L_E \neq \emptyset$: $(10, E, F)$ and $(10, E, D)$ are added to I with priority 3. $L_B \neq \emptyset$: $(8, B, A)$ is added to I with priority 2. L_D, L_F, L_A are updated accordingly. R_F and R_D are also updated whereas R_A is not ($\mathcal{F}_{SP}((4, 0, 0)) < \mathcal{F}_{SP}((4, 7, 8))$).
- $(+, 8, 10, D, F)$: A_D and A_E are updated. $(10, D, F)$ and $(10, F, D)$ are added

to I with priority 4. L_D,L_F, R_D and R_F are not updated ($\mathcal{Q}_{SP}((4,4,2)) < \mathcal{Q}_{SP}((4,8,3))$ and $\mathcal{F}_{SP}((4,4,2)) < \mathcal{F}_{SP}((4,8,3))$).

- The Algorithm return the distances from the source A: $d_A = 0$, $d_B = 1$, $d_C = 2$, $d_D = 3$, $d_E = 2$ and $d_F = 3$.

In Straph, the default implementation of optimal temporal paths is the single source *L-Algorithm*. Below we provide a snippet of code computing optimal temporal paths:

```
S = stream_graph()
source = 'A' # or (5,'A') for a temporal source node
S.distances(source) # Shortest Paths
S.latencies() # Fastest Paths
S.times_to_reach() # Foremost Paths
S.times_to_reach_and_lengths(source) # Shortest Foremost Paths
S.distances_and_durations(source) # Fastest Shortest Paths
S.latencies_and_lengths(source) # Shortest Fastest Paths

# A given path can be specified by inputting a destination
destination = 'B' # or (10,'B') for a temporal destination node
S.distances(source,destination)
```

Proofs and Complexities

Theorem 5.3.1. *Given a stream graph $S = (T, V, W, E)$, a path objective function $\mathcal{F}$ and an associated domination function $\mathcal{Q}$, the L-Algorithm computes optimal temporal paths properties, according to $\mathcal{F}$, from a temporal source node $(t, x) \in W$ to every other nodes $v \in V$ in $O(\Omega \cdot mlog(m) + M + N)$ time and $O(n + m)$ space.*

Proof. **Correctness** The loop invariant for Algorithm 2 is the following: at the end of one loop iteration, for time c and for each vertex v we have computed:

- the triplet corresponding to the optimal path from (t, x) to (c, v), stored in R_v;
- the triplet corresponding to the path that minimizes Q among all paths from (t, x) to v's current node segment, stored in L_v.

For a given time c, one iteration of the loop of Algorithm 2 adds to the current graph all the links that start at c. As all links that disappear are removed at the end of the loop, this means that the current graph A contains exactly the links that exist at time c.

We will now show that Algorithm 3 currently computes the values of L and R for time c, which will show that the loop invariant for Algorithm 2 holds. Notice that values of L_v and/or R_v can change for node v only if a better path than those observed before for v's current node segment arrives at time c on node v (or if a path arrives at v and no path was observed before). This can happen for one of two reasons:

- either the path starts from x at time c, or

- the path arrives to some node w before time c, then goes from w to v by going through one or several links at time c.

Algorithm 3 will then explore all paths involving one of these links and possibly several other links at time t. It starts with I containing the links starting at time c (in both directions), and the links from x existing at time c. Therefore, for any vertex v, the first link occurring at time c on the path from (t, x) to (c, v) is in I .

Let C be the set of nodes v such that either v is the origin of a link placed in I in Algorithm 2, or such that the value of L_v must be updated, i.e. there is a path from (t, x) to (c, v) that has a better value of the domination function, $\mathcal{Q}$, than any path from (t, x) to (c', v), where c' is the event time before c. We will show that the following loop invariant holds: (1) at the beginning iteration, I contains all starting links, together with links of which one extremity is in C and has been observed, minus the links that have already been dealt with; (2) the priority associated to each link (e, w, y) corresponds to the quality of the path going from (t, x) to (c, w) currently stored in L_w, to which the link (c, w, y) has been added.

This is true at the beginning of the first iteration. In each iteration, one link (e, w, y) is removed from I and considered. This is the link with minimum priority, i.e. it corresponds to the path (triplet) with minimum Q value among all paths that are yet to explore. We compare the domination function of the corresponding path to y to the triplet stored in L_y. If this path is better than one of them, we upgrade the corresponding values. Moreover, if the domination function value for y is updated this may correspond to better paths to neighbours h of y, therefore we add links (e, y, h) to I. It is easy to see that this preserves the loop invariant.

This, together with the observation that all nodes in C have necessarily a neighbour that is in C or that corresponds to a link initially in I, shows that values of L are correctly computed for each event time.

Finally, by definition of a domination function, we know that it is only necessary to store the path that minimizes the domination function in order to obtain the longer paths that will be optimal for the objective function. This, together with the observation that R is correctly updated each time a path is explored, completes the proof of correctness.

(**Complexity**) Algorithm 3 is in $O(mlog(m))$. There can be up to $O(m)$ elements in I as each link is visited only once. Inserting (or popping) an element in (of) I, a priority queue, is in $O(log(m))$ (lines 4 and 16). All executions of line 14 take a total time in $O(m)$ as there can be up to m ongoing temporal links at a given time.
The initialisation of algorithm 2 takes $O(n)$ (line 4-9) then the algorithm proceeds to scan every link arrivals and departures as well as node departures in S: $M + N$ elements (lines 12,24 and 26). Inserting or removing an element in A is in $O(n)$ (lines 13 and 25). Inserting links in I is in $O(mlog(m))$ (lines 16,19,21). The "Dijkstra's procedure", algorithm 3 (line 23), will be called at most Ω times (lines 10 and 23). Other steps are in $O(1)$. Thus a total time complexity in $O(\Omega \cdot mlog(m) + M + N)$.

The space complexity resides in the size of A, I, L and R. The size of A and I are in $O(m)$ and those of L and R in $O(n)$. Thus a space complexity in $O(n+m)$.

□

Observations and Remarks

Observation 5.3.1. *The L-Algorithm can be adapted to compute optimal temporal paths from a source node $x \in V$ (rather than a temporal node (t,x)) to all the other nodes in V. We only need to modify the initialisation of L_x (lines 4 and 5): $\forall[\beta_x,\theta_x] \in \overline{T_x}$ we add $(\beta_x,\theta_x,0)$ to L_x and insert any of these elements in R_x. This modification of the initialisation is in $O(N)$ and does not impact the time and space complexities of L-Algorithm.*

Observation 5.3.2. *Let us consider a stream graph, $S=(T,V,W,E)$, equivalent to a static graph, i.e. where $T=\{t\}$. Then the L-Algorithm can be assimilated to a version of Dijkstra's algorithm where elements of the priority queue are links instead of nodes. Indeed, the only parameter subject to change, in the triplets of L, would be the distance d (as $a=s=t$). Only shortest paths are considered and the complexity is reduced to $O(mlog(m)+n)$ as $\Omega=1$, $n=N$ and $m=M$.*

Observation 5.3.3. *Let us consider temporal nodes continuously present, $n=N$, and temporal links without duration, as in the case studied by Wu et al. [109]: $\forall l \in E^*$, $l=(b,b,u,v)$. Let us apply the L-Algorithm on such a stream graph. We can distinguish two cases:*
Link arrivals are distinct, $M \leq \Omega$, and at any instant $t \in T$ the induced static graph G_t contains at most two nodes and one link. Each temporal link is processed independently, we do not need I to be a priority queue as I contains at most one element. The time complexity is reduced to $O(M+n)$. Otherwise, the algorithm of Wu et al. [109] cannot be applied on a stream graph with simultaneous link arrivals. Indeed, Wu et al. make the assumption that the traversal time of each temporal link is strictly positive: if there exists simultaneous temporal links only one of them will be present in a temporal path as such links cannot be concatenated to one another.

Remark 5.3.1. *The following example shows why it is necessary to add ongoing links containing the source (lines 20-21, algorithm 2). Let us consider the stream graph in Figure 5-3, and apply the L-Algorithm for the shortest fastest path problem, $\mathcal{F}_{SFP} = (\mathcal{A}(\cdot)-\mathcal{S}(\cdot),\mathcal{D}(\cdot))$, $\mathcal{Q}_{SFP}=(-\mathcal{S}(\cdot),\mathcal{D}(\cdot))$, on the source node X. The shortest fastest paths from X to E, F and G are, respectively, $P_E=(4,X,D)$, $(4,D,E)$, $P_F=(6,X,C)$, $(6,C,D)$, $(6,D,F)$ and $P_G=(8,X,B)$, $(8,B,A)$, $(8,A,D)$, $(8,D,G)$.*

At instant $t=3$ the algorithm has already browsed $(0,4,X,D)$, $(0,8,X,B)$, $(1,6,X,C)$, $(0,8,B,A)$, $(2,6,D,C)$ and $(3,8,,A,D)$. L_D contains the canonical triplet $(4,0,1)$ corresponding to the path from X to D taking the link $(0,4,X,D)$. This triplet will be used at time 4 to obtain the shortest fastest path to E - more precisely the canonical triplet corresponding to this path: $(\mathcal{A}(P_E),\mathcal{S}(P_E),\mathcal{D}(P_E))=(4,4,2)$.

However, if triplets are only updated upon link arrivals without adding ongoing links

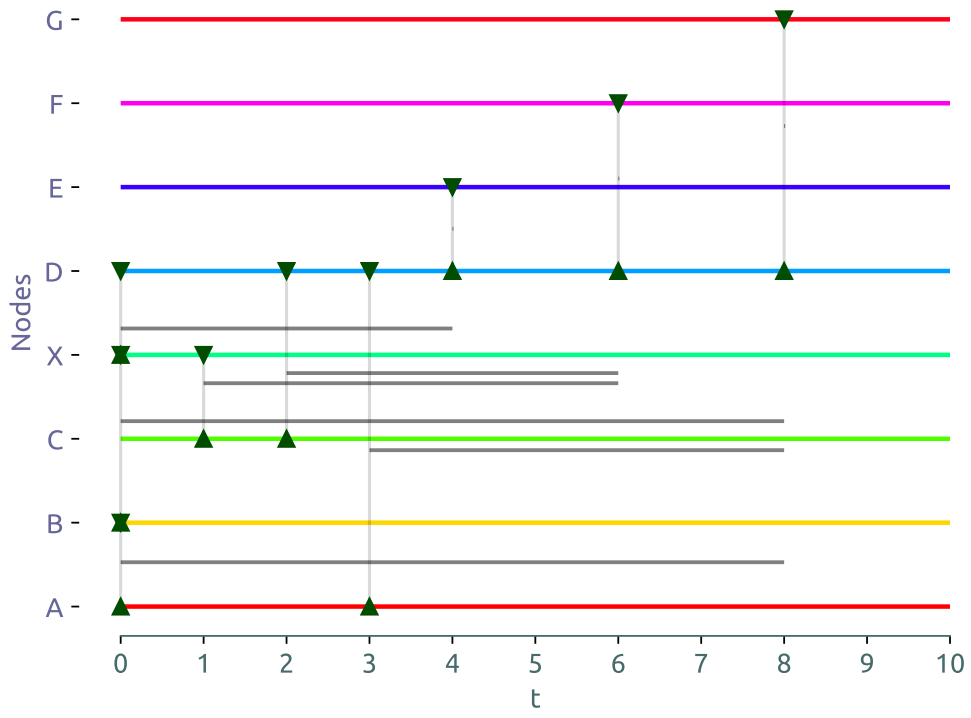

Figure 5-3: Illustration of shortest fastest paths complexity

containing the source, X, then the shortest fastest paths from X to nodes F and G will use the link $(0, 4, X, D)$ as $L_D = (4, 0, 1)$. The obtained triplets will be $R_F = (4, 6, 2)$ and $R_G = (4, 8, 2)$ and the corresponding paths will have a duration of 2 and 4. As P_F and P_G both have a null duration, these triplets do not correspond to shortest fastest paths. Therefore the need to consider ongoing links containing the source at times 6 and 8 to correctly maintain the canonical triplet in L_D (respectively $L_D = (6, 6, 2)$ at time 6 and $L_D = (8, 8, 3)$ at time 8).

5.4 Condensation Based Algorithms

In this section, we show that we can use connectivity properties of the condensation representation of a stream graph, introduced in chapter 4, to design efficient algorithms for the foremost and fastest path problems.

5.4.1 Time to reach and foremost paths

In the following we introduce condensation based algorithms to compute time to reach and foremost path. These algorithms exploit the particular structure and properties of the condensation of a stream graph, a directed acyclic graph. They derive from classical algorithms such as breadth first search (BFS). Nevertheless their implemen-

tations can be difficult and have their subtleties. Hence, in the following, we will detail several steps relating to pitfalls or technical improvements.

In Chapter 4 we have shown that paths in a stream graph can be represented by paths in its condensation, $G_{\mathscr{C}}$. Instead of returning a foremost path in S between (t, u) and v (or (t', v)) our algorithm will return a corresponding path in $G_{\mathscr{C}}$ between $C(t, u)$ and $C(v)$ (or $C(t', v)$).

Time to Reach and Arbitrary Foremost Path:

Algorithm 4 computes the time to reach a node v from a temporal node (t, u) as well as the corresponding condensation path in $G_{\mathscr{C}}$. The algorithm explores $G_{\mathscr{C}}$ in a BFS manner. It starts at the component containing the *temporal source node*. Then, whenever the BFS reaches a component containing the *destination node*, $(I, X) \in \mathscr{C}$ such that $v \in X$, the current path is stored as well as the potential time to reach $min(I) - t$ as a possible solution. When the algorithm cannot explore further, having visited all the components containing the *destination node*, we output the condensation path, a foremost path, as well as the time to reach.

This algorithm can easily be improved: the current time to reach can be used as a threshold. Indeed if the (potential) time to reach corresponding to a condensation path exceeds a previous solution, this path cannot be a foremost path. This threshold will discard any longer path and prevents the unnecessary exploration of several parts of $G_{\mathscr{C}}$. We present a detailed pseudo-code, including this improvement, in algorithm 4.

Remark 5.4.1. *A depth first search (DFS) could have been used rather than a BFS without altering the algorithm's complexity. However, as a stream graph can span a long time period, we do not want to explore the condensation in its temporal dimension, its "length" in the graphical representation (as with a DFS) but rather in its topological dimension, its "height" in the graphical representation (as in a BFS). A BFS will, in priority, explore nodes with an earlier begin time, which a DFS will not. In doing so, we can set a tight threshold in a quicker way, resulting in a tangible amelioration in practice.*

Remark 5.4.2. *If we want to adapt Algorithm 4 to compute $T_t(u, (t', v))$ and a foremost path from $C(t, u)$ to $C(t', v)$ instead of $T_t(u, v)$ and a foremost path from $C(t, u)$ to $C(v)$; we have to add the following condition in the test line* 12*: we need to assert that $[b, t'] \in T_v$.*

Theorem 5.4.1. *Algorithm 4 computes the time to reach $v \in V$ from a given $C(t, u) \in \mathscr{C}$ and a corresponding foremost path in $G_{\mathscr{C}}$ in $O(m_c + n_c)$. Its space complexity is $O(n_c \cdot n + m_c)$.*

Proof. The algorithm browses $G_{\mathscr{C}}$ starting from $C(t, u)$. We suppose that there exists a solution. The test line 12 ensures the algorithm to find $C(t', v) \in C(v)$, such that $t' - t$ is the time to reach $T_t(u, v)$. If there is no solution the algorithm will visit all components (line 9 and 20) and output an empty path (line 5) and a time to reach

Algorithm 4 Time to Reach $T_t(u, v)$ and a Foremost Path from $C(t, u)$ to $C(v)$ in $G_{\mathscr{C}}$.

Def: $foremost\_path(G_{\mathscr{C}}, t, C(t, u), v)$
Input: The condensation $G_{\mathscr{C}}$ of a stream $S = (T, V, W, E)$, $t \in T$, $C(t, u) \in \mathscr{C}$ and $v \in V$.
Output: A foremost path from $C(t, u)$ to $C(v)$ in $G_{\mathscr{C}}$ (empty if unreachable) and $T_t(u, v)$ (infinite if unreachable).
Mark all components in $\mathscr{C}$ as unseen
$foremost\_path \leftarrow \emptyset$
$ttr = \infty$ ▷ Initial time to reach
$C_u = C(t, u)$ ▷ The component containing the *source* (t, u)
$path\_queue = [[C_u]]$ ▷ Queue containing current paths
while $path\_queue$ is not empty **do**
    $P = path\_queue$.pop()
    Let $C = ([b, e], X)$ be the last element of P
    if $v \in X$**and** $b < ttr + t$ **then**
        $ttr = max(b, t) - t$
        $foremost\_path = P$
    for all C' following C in $G_{\mathscr{C}}$ **do**
        $([b', e'], X') = C'$
        if C' is not seen **and** $b' < ttr + t$ **then**
            Add C' to the current path P to form P'
            Mark C' as seen.
            Add P' to $path\_queue$.
return $foremost\_path, ttr$

equal to infinity (line 6).
The time complexity of this algorithm is one of a BFS: $O(m_c+n_c)$ with n_c the number of nodes of $G_{\mathscr{C}}$ and m_c its number of links. The space complexity is in $O(n_c \cdot n + m_c)$ as each condensation node, a strongly connected component, can contain up to n nodes and the maximal length of a path in $G_{\mathscr{C}}$ is bounded by m_c. □

We present a second algorithm to compute the time to reach $\mathcal{T}_t(u, v)$. It consists in a traversal of the condensation of S in a "Dijkstra" manner. The main difference with Algorithm 4 is that we keep a time ordering on the nodes of $G_{\mathscr{C}}$, in order, when we choose a neighbour, to pick $C = (I, X) \in \mathscr{C}$ such that $min(I)$ is as small as possible. The motivation, to choose a time ordering, can be seen as a way to set a tight threshold the fastest as possible. Indeed, if the algorithm explores in priority components that ends the earliest, it will probably find a foremost path whenever it reaches a component containing the *destination node*.

Algorithm 5 Time to reach v from a given temporal node (t, u).

1: **Def:** $time\_to\_reach(G_{\mathscr{C}}, t, C(t, u), v)$
2: **Input:** The condensation $G_{\mathscr{C}}$ of a stream $S = (T, V, W, E)$, $t \in T$ $(t, u) \in W$ and $v \in V$.
3: **Output:** The time to reach $T_t(u, v)$ (infinity if unreachable).
4: Set Q to empty priority queue
5: Mark all components as unseen
6: $([b, e], X) = C(t, u)$ ▷ The component containing the *source*
7: Q.add($([b, e], X)$) with key t
8: **while** Q is not empty **do**
9: Pop C, the element with the smallest key, from Q
10: $([b, e], X) = C$
11: **if** $v \in X$ **then return** $max(b, t) - t$
12: **for** all C' following C **do**
13: **if** C' is not seen **then**
14: $([b', e'], X') = C'$
15: Mark C' as seen
16: Q.add(C') with key b'
return $infinity$

Theorem 5.4.2. *Algorithm 5 computes the time needed to reach a given node v from a given $C(t, u) \in \mathscr{C}$ in time $O(m_c log(m_c)+n_c)$. Its space complexity is $O(n_c \cdot n+m_c)$.*

Proof. The proof is the similar of the one of Algorithm 4. The time complexity is a bit different: the log factor comes from the use of a priority queue. The space complexity is identical. □

Remark 5.4.3. *In order to support path queries directly from stream nodes we need to build the following index: the index associating a stream node to the connected*

components containing it in $G_{\mathscr{C}}$, $\mathcal{I} : u \to C(u)$. The initialisations of algorithms 4 and 5 only necessitate to browse every component in $C(u)$ - in order to find $C(t, u)$ - which can be done in $O(n_c)$. The space complexity of such an index is in $O(n_c \cdot n)$. Hence, we can support queries with stream nodes as inputs without altering the complexities of the considered algorithms.

Count and enumerate all Foremost Paths:

Computing all foremost paths in the condensation graph is a reduction of a classical problem, which is the enumeration of all possible paths between a source - component containing the *temporal source node* - and a destination - component containing the *destination node* - in a graph. This problem is NP-Hard [81].

In a directed acyclic graph this problem becomes simpler. We can count the number of foremost paths using dynamic programming in $O(n_c+m_c)$. However as the number of paths can be exponential we cannot hope for a polynomial algorithm to enumerate them, as each output will cost at least $O(1)$.

We propose an algorithm to enumerate all foremost paths from a temporal node $(t, u) \in W$ to a node $v \in V$: Algorithm 6. It proceeds as follows: given a starting time and a *source* the algorithm proceeds to explore each node of the condensation graph in the manner of a breadth first search (BFS), except that we do not keep a register of 'seen' components. Once it has found a component that contains the destination, the algorithm uses the current time to reach as a threshold in order for the algorithm to discard any longer path, meanwhile all possible paths corresponding to this time to reach are registered. After completion the algorithm outputs all condensation foremost paths from $C(t, u)$ to $C(v)$ as well as the time to reach v from (t, u).

If we only desire the number of foremost paths, a simpler version can be done by replacing $foremost\_paths$ with a counter. This counter would be initialised to 0 line 4, reseted to 1 lines $12 - 14$ and incremented line 16.

In Chapter 4 we have shown that a condensation path can encode an infinite number of stream paths. The number of distinct foremost paths in $G_{\mathscr{C}}$ is finite. Therefore, we argue that algorithm 6, by returning all foremost paths from a source to a destination in $G_{\mathscr{C}}$, can quantify the amount of foremost paths in a stream graph. Whether by counting the number of distinct paths in $G_{\mathscr{C}}$ or by computing the total volume of the components involved in these paths (we refer to [59] for detailed explanations regarding the volume of temporal paths in stream graphs). This quantification can be really helpful to compute particular centrality measures.

The worst-case complexity of algorithm 6 is exponential as we may have to explore every possible path in $G_{\mathscr{C}}$. However, in real-world datasets, stream graph condensations have a particular property (cf section 4.1.5, Table 4.1): their mean out-degree, d_c, is really low, typically inferior to 1. Hence, the exponential growth of the number of paths in the condensation is limited, allowing the algorithm to scale. The employed

Algorithm 6 All Foremost Paths from $C(t,u)$ to $C(v)$.

Def: $all\_foremost\_paths(G_{\mathscr{C}}, t, C(t,u), v)$
Input: The condensation $G_{\mathscr{C}}$ of a stream $S = (T, V, W, E)$, $t \in T$, $C(t,u) \in \mathscr{C}$ and destination node $v \in V$.
Output: All Foremost path from $C(t,u)$ to $C(v)$ in $G_{\mathscr{C}}$ (empty if unreachable) and $T_t(u,v)$ (infinity if unreachable).
$foremost\_paths$ is an empty list
$ttr = infinity$ is the time to reach
$([b,e],X) = C(t,u)$ ▷ Component containing the *source*
$path\_queue = [[([b,e],X)]]$ ▷ Queue containing current paths
while $path\_queue$ is not empty **do**
    $P = path\_queue$.pop()
    $([b,e],X) = C$
    if $v \in X$ **and** $b < ttr + t$ **then**
        $ttr = max(b,t) - t$
        Set $foremost\_paths$ to empty
        Add P to $foremost\_paths$
    else if $v \in X$ **and** $max(b,t) == ttr + t$ **then**
        Add P to $foremost\_paths$
    for all C' following C in $G_{\mathscr{C}}$ **do**
        $([b',e'],X') = C'$
        if $b' \leq ttr$ **then**
            Add C' to P to form P'
            Add P' to $path\_queue$
return $foremost\_paths, ttr$

strategy takes advantage of the structure and properties of $G_{\mathscr{C}}$ resulting in an efficient algorithm to tackle real world datasets.

Below we provide a snippet of code, using *Straph*, to compute a time to reach using the condensation graph:

```
S = stream_graph()
cdag = S.condensation_dag()
source = 'A' # or (5,'A') for a temporal source node
destination = 'B'
cdag.time_to_reach(source,destination)
```

5.4.2 Latency and fastest paths

Similarly as in the previous section, we design algorithms for the computation of latency and fastest path using $G_{\mathscr{C}}$.

Latency and Arbitrary Fastest Path:

Algorithm 7 computes a fastest path in $G_{\mathscr{C}}$ from $C(u)$ to $C(v)$. The main steps are:

1. All components containing the *source* are stored in a set: L. The algorithm starts from $C_0 = (I_0, X_0) \in C(u)$ with $min(I_0)$ as low as possible (the first component where u appears). Then it performs a *foremost path* (Algorithm 4) procedure with parameters $(G_{\mathscr{C}}, max(I_0), C_0, v)$ to reach $C_k \in C(v)$, obtaining $P = (C_0, \ldots, C_k)$.
2. All components $C_i \in P$ containing u are removed from L, and consider as *seen*. We denote by C_u the latest one in P. The associated duration between $C_u = (I_u, X_u)$ and $C_k = (I_k, X_k)$, $min(I_k) - max(I_u)$ is kept as the current latency, if it is lower than previously defined.
3. For each *unseen* $C \in C(u)$ from L we perform steps (1) and (2).

Proposition 5.4.1. *Algorithm 7 computes the latency and a corresponding fastest path in $G_{\mathscr{C}}$ from $C(u)$ to $C(v)$ in time $O(n_c + m_c)$. Its space complexity is in $O(n_c \cdot n + m_c)$.*

Proof. The proof of Algorithm 7 is similar to the one of Algorithm 4. The procedure browses every components containing the source u. We consider every foremost paths from $C(u)$ to $C(v)$ and only keep the one with the latest departure time. As we maximize the departure time from u over the set of paths arriving the earliest in v, the resulting duration is the latency and the corresponding path is a fastest path in $G_{\mathscr{C}}$.

The time complexity of all *foremost_path* procedures (line 9) is in $O(n_c + m_c)$. Indeed, if a *foremost_path* procedure reaches worst case complexity, $G_{\mathscr{C}}$ is browsed entirely, consequently every component in $C(u)$ has been considered. Otherwise

Algorithm 7 Latency $\ell(u, v)$ and an arbitrary fastest path from $C(u)$ to $C(v)$ in $G_{\mathscr{C}}$.

Def: $fastest\_path(G_{\mathscr{C}}, C(u), v)$
Input: The condensation graph $G_{\mathscr{C}}$ of a stream $S = (T, V, W, E)$, $C(u) \subseteq \mathscr{C}$ and a destination node v in V.
Output: A fastest path from $C(u)$ to $C(v)$ in $G_{\mathscr{C}}$ and the latency $\ell(u, v)$.
Add all elements $(I, X) \in C(u)$ to L. L is ordered temporally in increasing order by $max(I)$.
$latency = \infty$
$fastest\_path \leftarrow \emptyset$
while L **is not** empty **do**
    Pop the first element of L: $C = (I, X)$
    $P, ttr = foremost\_path(G_{\mathscr{C}}, max(I), C, v)$
    if $ttr \neq \infty$ **then**
        for C_i **in** P **do**
            $(I_i, X_i) = C_i$
            if u in X **then**
                $C_u = C_i$
                Remove C_i from L
        $(I_u, X_u) = C_u$
        Get $C_k = (I_k, X_k)$ the latest component in P
        $ttr = min(I_k) - max(I_u)$
        if $ttr < latency$ **then**
            $latency = ttr$
            $(C_0, \ldots, C_k) = P$
            $fastest\_path = (C_u, \ldots, C_k)$
return $fastest\_path$, $latency$

each *foremost_path* procedure has browsed a part of the graph that will never be browsed again: all components containing u are removed from L, line 15, hence no *foremost_path* procedure will be run from these components. All node and link of $G_{\mathscr{C}}$ have been visited only once during all *foremost path* procedures, amounting to $O(n_c + m_c)$.
The space complexity is the same as in Algorithm 4. □

Observation 5.4.1. *Algorithm 7 can be employed to efficiently answer reachability queries between nodes in a stream graph. As shown in chapter 4, if a node v is reachable from a node u then there exists a path from $C(u)$ to $C(v)$ in $G_{\mathscr{C}}$. Algorithm 7 tackles this problem, if it outputs a condensation path (or a finite latency) the query is answered positively otherwise an empty path (or an infinite latency) means that v is not reachable from u. This can be done by defining an index as in remark 5.4.3.*

As in the previous section one may want to count or enumerate all fastest paths. Algorithm 8 tackle this problem.

Remark 5.4.4. *A hybrid algorithm computing Shortest Fastest Paths can be built combining Algorithm 7 with the L-Algorithm (Algorithm 2). Indeed if the L-Algorithm is applied on the results of Algorithm 7 - which can be represented as substreams of the initial stream graph - it's easy to obtain an algorithm to compute shortest fastest paths.*
However, in practice, such an algorithm is far slower than the L-Algorithm for SFP.

Below we provide a snippet of code computing latencies using the condensation graph

```
S = stream_graph()
cdag = S.condensation_dag()
source = 'A'
destination = 'B'
cdag.latency(source,destination) # Duration of Foremost Paths
```

5.5 Experiments

We have evaluated the performances of the *L-Algorithm* and condensation based algorithms on 14 real world stream graphs (see Chapter 1.3). As our datasets are too important to compute every optimal temporal path between every node pairwise, we choose to follow the evaluation protocol described in [109]. Firstly, the algorithms are evaluated for 100 nodes randomly picked and secondly on the 10 nodes with the highest instant degree.

As shown in Figure 5-4, we can differentiate three types of path problems. Foremost path and shortest foremost path problems are quicker to compute, as expected. Indeed, as the starting time is fixed, we only consider the node segment containing the starting time as a potential source and most of the time we do not have to browse every temporal link in the stream graph. The second type of temporal path problems

Algorithm 8 Compute all fastest paths from $C(u)$ to $C(v)$ in $G_{\mathscr{C}}$

Def: $all\_fastest\_paths(G_{\mathscr{C}}, C(u), v)$
Input: The condensation graph $G_{\mathscr{C}}$ of a stream $S = (T, V, W, E)$, $C(u) \subseteq \mathscr{C}$ and a destination node v in V.
Output: All fastest paths from $C(u)$ to $C(v)$ in $G_{\mathscr{C}}$ (empty if unreachable) and $\ell(u, v)$ (infinity is unreachable).
Add all elements $(I, X) \in C(u)$ to L. L is ordered temporally in increasing order by $max(I)$.
$latency = infinity$
$fastest\_paths$ is empty
while L is not empty **do**
    Pop the first element of L: $C = (I, X)$.
    $Q, ttr = all\_foremost\_paths(G_{\mathscr{C}}, max(I), C, v)$
    if $ttr \neq infinity$ **then**
        for P in Q **do**
            for C_i in P **do**
                $(I, X) = C_i$
                if u in X **then**
                    $C_u = C_i$
                    Remove C_i from L.
            $(I_u, X_u) = C_u$
            $(C_0,\ldots,C_k) = P$
            $(I_k, X_k) = C_k$
            $ttr = min(I_k) - max(I_u)$
            if $ttr < latency$ **then**
                $latency = ttr$
                $fastest\_paths = [(C_u,\ldots,C_k)]$
            else if $ttr - max(I_u) == latency$ **then**
                $fastest\_paths.append((C_u,\ldots,C_k))$
return $fastest\_paths$, $latency$

	FoP	SP	FP	SFoP	SFP	FSP
UC	9.3e-5	1.4e-4	1.1e-4	9.3e-5	-	1.4e-4
HS 2012	7.14e-4	1.1e-3	1.1e-3	7.2e-4	1.6e-3	1.14e-3
Digg	1.9e-5	2.3e-5	2.4e-5	1.9e-5	2.8e-5	2.3e-5
Infectious	4.6e-5	5.4e-5	5.3e-5	4.6e-5	5.6e-5	5.4e-5
Twitter	1.2e-5	1.3e-5	1.4e-5	1.3e-5	1.5e-5	1.5e-5
Linux	1.3e-4	1.5e-4	1.5e-4	1.4e-4	1.8e-4	1.6e-4
Facebook	1.3e-4	1.5e-4	1.5e-4	1.3e-4	1.6e-4	1.5e-1
Epinions	4.8e-5	5.2e-5	6.2e-5	5.2e-5	9.4e-4	5.8e-5
Amazon	2.2e-5	2.6e-5	2.6e-5	2.2e-5	2.7e-5	2.6e-5
Youtube	1.9e-5	2.7e-5	2.8e-5	2.1e-5	-	2.9e-5
Movielens	9.8e-4	1.3e-3	1.3e-3	9.7e-4	1.7e-2	1.3e-3
Wiki	6.2e-5	7.5e-5	7.7e-5	6.3e-5	9.2e-5	7.7e-5
Mawilab	3.2e-4	3.4e-4	3.2e-4	2.9e-4	-	3.2e-4
Stackoverflow	1.6e-4	1.9e-4	1.9e-4	1.6e-4	1.5e-3	1.9e-4

Table 5.2: L-Algorithm running times in seconds (random sources) (missing values correspond to the timeout of a procedure, set to 15000s)

contains the SP, FP and FSP problems, which have similar running times. Finally, the SFP problem stands out as its computation time can be larger than for the other path problems. This can be explained by the fact that many paths can improve its domination function during the browsing of temporal links, indeed there can be many instantaneous paths (fastest paths) between two nodes resulting in many more calls to the *Dijktra_Update* procedure (see Algorithm 3).

In Figure 5-5 we present performances of the *L-Algorithm* (Algorithm 2) and condensation based algorithms (Algorithms 4 7) for the foremost and fastest path problem. In most of the datasets condensation based algorithms are quicker. This observation is coherent with the algorithms theoretical complexities. However, we point out that the preprocessing time needed to compute the considered condensations can be important (see Chapter 4).

For the sake of completeness, we present in Table 5.2 running times of the *L-algorithm*, in seconds, computing an optimal temporal path for the foremost path, shortest path, fastest path, shortest foremost path, shortest fastest path and fastest shortest path problems in the considered datasets. In Table 5.3 we present the same results from high degree source nodes. In Tables 5.4 (random sources) and Tables 5.5 (high degree sources) we present the running times of the condensation algorithms for the foremost and fastest path problems and we compare them to the running times of *L-Algorithm*.

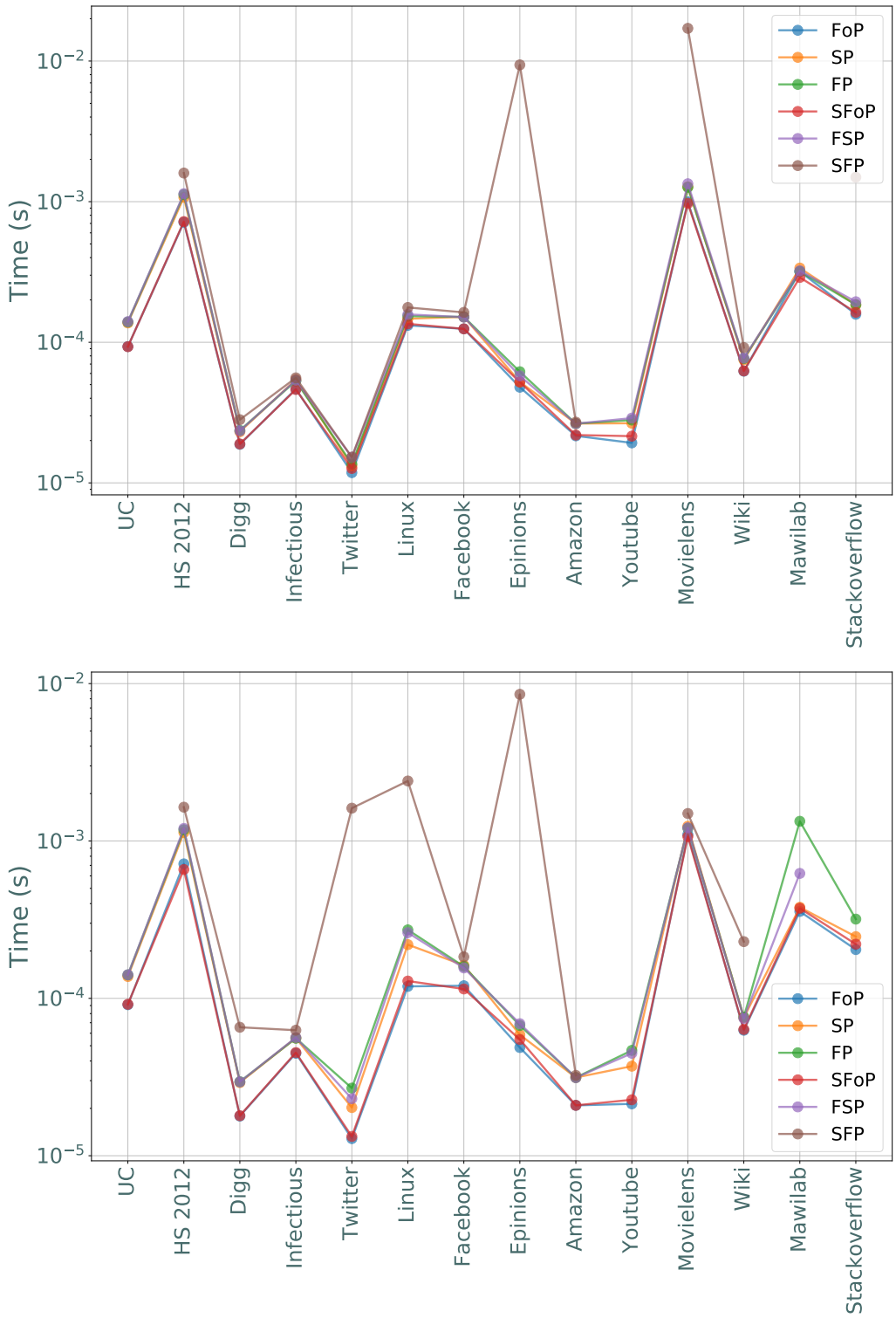

Figure 5-4: Running Times (s) of the *L-Algorithm* for the foremost, shortest fastest, shortest foremost, fastest shortest and shortest fastest path problems (top: random sources, bottom: high degree sources).

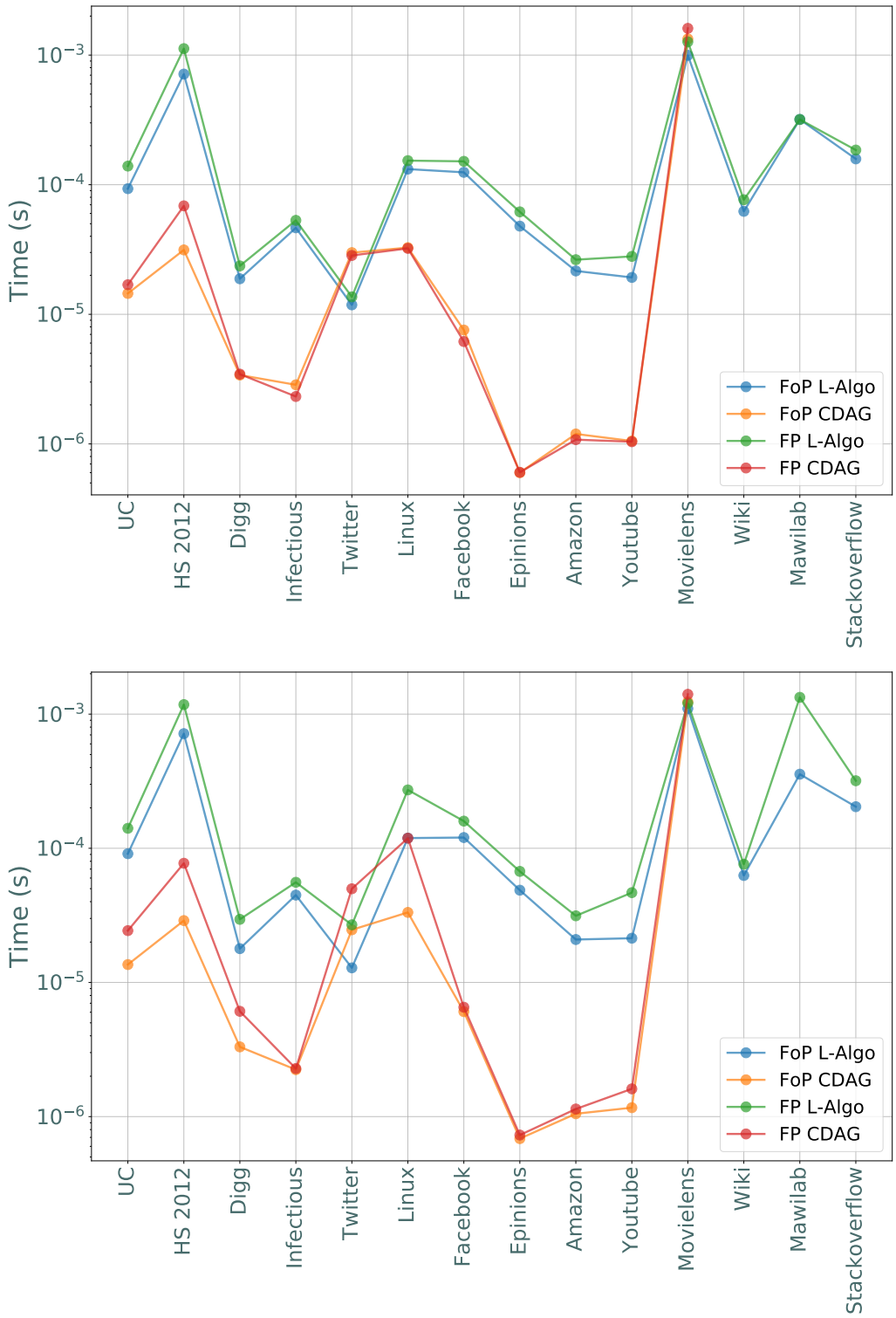

Figure 5-5: Running Times (s) of the *L-Algorithm* and of condensation algorithms for the foremost and fastest path problems (top: random sources, bottom: high degree sources).

	FoP	SP	FP	SFoP	SFP	FSP
UC	9.1e-5	0.000138	0.000141	9.2e-5	-	0.000141
HS 2012	0.000715	0.001132	0.001178	0.000659	0.00164	0.001201
Digg	1.8e-5	2.9e-5	3e-5	1.8e-5	6.6e-5	2.9e-5
Infectious	4.5e-5	5.6e-5	5.6e-5	4.5e-5	6.3e-5	5.6e-5
Twitter	1.3e-5	2e-5	2.7e-5	1.3e-5	0.001616	2.3e-5
Linux	0.000119	0.000219	0.000272	0.000129	0.002405	0.000261
Facebook	0.00012	0.000162	0.000159	0.000115	0.000184	0.000156
Epinions	4.9e-5	5.9e-5	6.7e-5	5.5e-5	0.00855	6.9e-5
Amazon	2.1e-5	3.1e-5	3.1e-5	2.1e-5	3.2e-5	3.1e-5
Youtube	2.1e-5	3.7e-5	4.7e-5	2.3e-5	-	4.5e-5
Movielens	0.001099	0.00124	0.001204	0.001067	0.001495	0.001213
Wiki	6.3e-5	7.5e-5	7.6e-5	6.3e-5	0.000229	7.5e-5
Stackoverflow	0.000204	0.000247	0.000319	0.000221	-	-
Mawilab	0.000357	0.000379	0.001334	0.000374	-	0.000622

Table 5.3: L-Algorithm running times in seconds (high degree sources) (missing values correspond to the timeout of a procedure, set to 15000s)

	FoP L-Algo	FoP CDAG	FP L-Algo	FP CDAG
UC	9.3e-5	**1.4e-5**	0.000139	**1.7e-5**
HS 2012	0.000714	**3.1e-5**	0.001122	**6.9e-5**
Digg	1.9e-5	**3e-6**	2.4e-5	**3e-6**
Infectious	4.6e-5	**3e-6**	5.3e-5	**2e-6**
Twitter	**1.2e-5**	3e-5	**1.4e-5**	2.8e-5
Linux	0.000132	**3.3e-5**	0.000153	**3.2e-5**
Facebook	0.000125	**8e-6**	0.000151	**6e-6**
Epinions	4.8e-5	**1e-6**	6.2e-5	**1e-6**
Amazon	2.2e-5	**1e-6**	2.6e-5	**1e-6**
Youtube	1.9e-5	**1e-6**	2.8e-5	**1e-6**
Movielens	**0.000996**	0.001334	**0.001266**	0.001609

Table 5.4: L-Algorithm and condensation algorithms running times in seconds (random sources) (missing values correspond to the timeout of a procedure, set to 15000s)

	FoP L-Algo	FoP CDAG	FP L-Algo	FP CDAG
UC	9.1e-5	**1.4e-5**	0.000141	**2.4e-5**
HS 2012	0.000715	**2.9e-5**	0.001178	**7.7e-5**
Digg	1.8e-5	**3e-6**	3e-5	**6e-6**
Infectious	4.5e-5	**2e-6**	5.6e-5	**2e-6**
Twitter	**1.3e-5**	2.5e-5	**2.7e-5**	5e-5
Linux	0.000119	**3.3e-5**	0.000272	**0.000119**
Facebook	0.00012	**6e-6**	0.000159	**7e-6**
Epinions	4.9e-5	**1e-6**	6.7e-5	**1e-6**
Amazon	2.1e-5	**1e-6**	3.1e-5	**1e-6**
Youtube	2.1e-5	**1e-6**	4.7e-5	**2e-6**
Movielens	**0.001099**	0.001235	**0.001204**	0.001405

Table 5.5: L-Algorithm and condensation algorithms running times in seconds (high degree sources) (missing values correspond to the timeout of a procedure, set to $15000s$

5.6 Conclusion

In this chapter we proposed a generic polynomial algorithm - which can be seen as a generalisation of the famous Dijkstra's algorithm - and theoretical concepts allowing this single source algorithm, the *L-Algorithm*, to compute many paths problems in stream graphs. We have shown that a path problem can be seen as an optimisation problem based on two functions, the objective and domination functions. We provide these functions for the foremost, fastest, shortest path problems as well as for multi-criteria path problems (shortest foremost paths, shortest fastest paths, fastest shortest paths). The corresponding algorithms were evaluated on 14 real-world datasets of various sizes, asserting their practical efficiency on stream graphs with tens of millions of temporal links.

In a second time, we have designed linear paths algorithms using previously defined concept: the condensation of a stream graph. In practice we have shown that these algorithms are faster or equivalent than our *L-Algorithms* for the foremost and fastest paths problems at the expense of a polynomial preprocessing. However, as the processing time can be important, these condensation algorithms should only be used if the objective is to compute a great number of paths. Given these encouraging results we aim to provide others stream graph representation tailored to facilitate the computation of other path problems.

The *L-Algorithm* calls for many extensions. Indeed, it can easily be adapted for weighted stream graphs links but we wonder if it would be the case for a modelisation where links have delays (where taking a link would take us further in time). Supposing that the corresponding objective and domination functions exist, the *L-Algorithm* could be used for much more complex path problems.

In the future, we hope to provide indicators in order to decide in what circumstances would it be preferable to use a condensation algorithm over the *L-Algorithm*. These indicators could be based on connectivity properties or on the trade-off between the time taken to compute the condensation and the amount of time saved by using a condensation algorithm rather than the *L-Algorithm*.

Conclusion

Stream graphs accurately model many real-world data; they are adapted for large-scale datasets through streaming possibilities as well as compact data structures allowing the manipulation of large-scale datasets in main memory. They are particularly well suited to model data stemming from network traffic [107], mobility traces [101] and online social interactions [35]. Such data benefit from our approach, as (directed, weighted) stream graphs capture most of their features without resorting to any form of (aggregated or discretised) approximation.

However, while theoretical concepts are numerous, the work needed to achieve efficient algorithm designs and associated challenge remains important. Moreover, in order to conduct such real-world applications, it is crucial to design and implement a convenient software to efficiently compute the properties of large stream graphs. Work in this direction has been partially achieved in this thesis through the design of connectivity and paths algorithms as well as their implementations in *Straph*.

In the following sections, we provide an overview of the contributions and we discuss future research directions.

Summary of Contributions

Our research focused on what we consider building blocks necessary for further applications and discoveries. This objective, requiring a good understanding of graph data structures and algorithms, is twofold:.

- Designing linear and polynomial algorithms for basic stream graphs computations, integrating and taking advantage of existing graph algorithms.
- Providing data structure to manipulate large-scale real-world data and evaluate stream graph algorithms in practice.

Our contributions fall within this framework: they either establish algorithmic results, or provide better data structures for handling stream graphs. Below we recall the main contributions obtained during this thesis:

- The most advanced open source python library for the manipulation, modeling,

analysis and visualisation of stream graphs: Straph (chapter 2)

- Efficient data structures to handle stream graphs (chapter 2)
- Two random stream graph generators based on the Erdős-Rényi and Barabási-Albert models (chapter 2)
- Algorithms to compute connectivity notions in stream graphs (chapter 3)
- Connectivity analysis of large scale real-world stream graphs (chapter 3)
- A connectivity based data representation of a stream graph which greatly reduces the complexity of computing reachability queries: the condensation of a stream graph (chapter 4)
- An alternative data representation paving the way to an efficient parallel framework for the computation of numerous properties on stream graphs: the stable directed acyclic graph of a stream graph (chapter 4)
- An approximation method speeding up numerous methods in practice while preserving the connectivity properties of a stream graph: the Δ-approximation (chapter 4)
- The first single source one-pass polynomial time algorithm computing all kinds of optimal temporal paths in stream graphs: the *L-algorithm* (chapter 5)
- Condensation based linear time algorithms for the computation of foremost and fastest paths (chapter 5)

Algorithmic contributions provide state-of-the-art algorithms for stream graphs and practical contributions demonstrate the effectiveness of this modeling. Whether these contributions stem from graph literature or are completely new, they scale to stream graphs with tens of millions of nodes and links.

Future Directions of Research

In the following we present future research directions stemming from our work. This presentation is ordered hierarchically by our own interpretation of their potential scientific and practical impact.

Global directions

We hope that the data structures presented in this dissertation could lead or inspire other data representations of stream graphs, the most promising one being the DAG ones in our opinion.

A major contribution would be the design of a global online framework allowing a real-time processing of stream graphs. As many practical cases would benefit from such a framework, for instance in a cyber-security context, detecting an abnormal

behaviour of a node or of a set of nodes should be instantaneous regarding the potentially catastrophic consequences of a cyber-attack. Likewise, detecting an infected person in a human contact network should be done as quickly as possible for obvious reasons. We hope to quantify how much these lossless stream graph properties bring to the table compared to graph properties on aggregated graphs or on a sequence of snapshots. The consequences from a machine learning perspective are promising. Statistically, a richer and unaltered information generally results in a better modeling.

As mentioned in chapter 4, many stream graphs concepts do not only provide "vertical" or "horizontal" slices but cover both dimensions at the same time. This approach should lead to better understanding of many phenomena in temporal data, especially in anomaly detection. Indeed an anomaly should not be restrained to a given set of nodes (vertical dimension) or a time window (horizontal dimension). A temporal anomaly may be composed of different nodes (and links) through time and for each node at a different time window.

Last but not least, one may notice that stream graphs are not only generalizations of graphs. They actually lie at the crossroad of two very rich and powerful scientific areas: graph theory and time series analysis. Integrating time series concepts to stream graphs has been left aside in our work but we hope to consider these aspects in the future.

Δ analysis

In chapter 4, we have presented an approximation scheme. We have evaluated and asserted its benefits on connectivity notions. However, this scheme needs to be evaluated in different contexts. We think that, as this scheme does greatly reduce the number of SCC in a stream graph, it would also reduce the number of stable connected components. The resulting stable DAG would also contain fewer nodes and the stable DAG framework should perform much faster. Likewise, the *L-Algorithm* proceeds to less costly procedures - *Dijkstra_Update* (see Chapter 5) - when the temporal links have the same arrival time, thus a shorter running time. However this calls for extended and thorough experimentations.

DAG extension

Our first exploration direction is a direct consequence of our work: try to investigate and quantify what stream graph metrics, computed using our algorithms - or with existing algorithms through the DAG framework - bring to the table in terms of understanding and modeling. Precisely, many properties could provide meaningful, in-depth information facilitating interpretations and analysis of such data.

It will be interesting to evaluate how our framework and its multiple variations through many graph algorithms would fare against other existing procedures. Under the assumption of an identical output, for instance an algorithm computing the nodes

core number over time should output a set of temporal nodes and their corresponding value; such a procedure would measure the practical interest of our framework without invalidating its overall possibilities.

Likewise, we have mentioned the possibility to design a streaming framework similar to the DAG parallel framework. An additional step consists in encoding the DAG differently: each stable connected component would be encoded accordingly to its direct predecessors in the DAG, meaning that only arrivals and departures of nodes/links would be encoded in each component. Such a framework would proceed in a streaming fashion on the stable DAG. It would start by computing properties on a component and then properties on its successors in the DAG, with a graph streaming algorithm, while taking into account the properties already computed on its direct predecessors. The obtained procedure should behave competitively against state-of-the-art. We can notice that this procedure leaves room for parallelization on each distinct weakly connected components - a connected stable DAG of its own - or on the stable DAG roots - stable connected components with a null in-degree in the stable DAG.

Then we should compare these two frameworks and consider when it is useful to resort to the streaming one. Indeed as many components (nodes of the condensation directed acyclic graph) possess only few nodes (see chapter 4) and as parallelization possibilities in the streaming framework are limited, it is unsure which framework will be most efficient in practice. Furthermore, this will allow us to decide for which properties it would be preferable to resort to a streaming procedure.

Different kind of DAG could be defined from different notions of connected components. A DAG could be designed for the computation of particular stream graph properties. Indeed as strongly connected components are, most of the time, stable connected components in practice (see chapter 4), for some properties it may be faster to run streaming algorithms in parallel on each SCC rather than classical algorithms on each stable connected component.

Another open question with countless applications consists in detecting communities efficiently. We hope to adapt, in the future, our DAG framework to provide community detection algorithms for stream graphs.

Path Algorithms

We are confident that our *L-Algorithm* (Chapter 5) could easily be adapted for directed and weighted stream graphs. However, the case of delayed links, where taking a link takes strictly positive time, remains an open question.
Another interesting track to explore would be the adaption of condensation algorithms provided in this dissertation to compute metrics specific to directed and/or weighted stream graphs. However, it would necessitate a notion of strongly connected components for directed stream graphs which does not exist at the time being.

Another track of improvement consists in exploring parallel implementations of the

L-Algorithm. It could be parallelized by running it independently on each weakly connected components. As computing WCC can be done linearly, the additional cost of computing WCC should be inferior to the gain of a parallel implementation.

The theoretical concepts of objective and domination functions generalize the one provided by Wu et al. [109] and in a particular way the one of Dijkstra - "a subpath of a shortest path being a shortest path" is a domination concept. We are considering which are the limits of this approach. Is there a path optimisation problem where no domination function could be provided? Are weighted stream graphs compatible with our procedure?
The improvement tracks are numerous as well as their potential applications.

Data Structures and Straph Extensions

The developed python library, *Straph*, contains implementations of all the algorithms, data structures and frameworks presented through this thesis. These implementations helped us to validate many results as well as analysing many real world datasets efficiently at scale. However, we are wondering as if our paradigms of developments and the corresponding implementations are well suited for an open source software. In the future, we aim to add many features and functionalities, specifically regarding the visualisation of stream graphs.

Random Stream Graph Generators

The random stream graph generators provided in chapter 2 must be compared to the state of the art and their practical behaviours need to be subject to a more extended evaluation. In particular, an important approach in graph studies consists in generating random graphs that have a prescribed set of properties. For instance, the Erdős-Rényi model generates graphs with prescribed size and density. Hence, in the future, we hope to provide theoretical proofs regarding their properties, such as the degree distribution, size and density of the generated stream graphs.

Final Statement

Whereas we have provided many elements towards a practical use of stream graphs, much remains to be done in the design of efficient algorithms, as well as the understanding of their complexity, for numerous theoretical concepts. We focused on basic graph theory concepts while more complex ones, such as betweenness centrality, remain to explore.
As mentioned, stream graph theory lies at the intersection of two scientific fields: graphs and time series. Within the frame of this thesis, we have not explored time series concepts. However, we consider this aspect to be important and we hope to provide advances in this direction, notably through our framework allowing the

computation of node and link time series corresponding to different kinds of time dependant properties (see chapter 4).

In this thesis, many problems have caught up our interest, particularly designing and implementing specific algorithms, data structures and representations for stream graphs. This has proven to be a very interesting and challenging research domain. Working throughout this dissertation was an exciting experience. The achieved work paves the way for numerous extensions towards the computation of more complex, finer grain concepts which will, without a doubt, bring a deeper understanding of these complex temporal structures.

Bibliography

[1] Amazon ratings network dataset – KONECT, April 2017.

[2] Digg network dataset – KONECT, April 2017.

[3] Epinions trust network dataset – KONECT, April 2017.

[4] Linux kernel mailing list replies network dataset – KONECT, April 2017.

[5] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. *ACM SIGMOD Record*, 18(2):253–262, June 1989.

[6] Eleni C. Akrida and Paul G. Spirakis. On Verifying and Maintaining Connectivity of Interval Temporal Networks. *Parallel Processing Letters*, 29(02), June 2019.

[7] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of modern physics*, 74, 2002.

[8] David Alberts, Giuseppe Cattaneo, and Giuseppe F. Italiano. An Empirical Study of Dynamic Graph Algorithms. *J. Exp. Algorithmics*, 2, January 1997.

[9] Albert-László Barabási et al. *Network science*. Cambridge university press, 2016.

[10] Alain Barrat and Ciro Cattuto. Temporal networks of face-to-face human interactions. In *Temporal Networks*, pages 191–216. Springer, 2013.

[11] V. Batagelj and M. Zaversnik. An O(m) Algorithm for Cores Decomposition of Networks. *arXiv:cs/0310049*, October 2003. arXiv: cs/0310049.

[12] Vladimir Batagelj and Selena Praprotnik. An algebraic approach to temporal network analysis based on temporal quantities. *Social Network Analysis and Mining*, 6(1):28, May 2016.

[13] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[14] Claude Berge. *The Theory of Graphs and Its Applications*. 1962.

[15] Jonathan Berry, Matthew Oster, Cynthia A. Phillips, Steven Plimpton, and Timothy M. Shead. Maintaining connected components for infinite graph streams. In *Proceedings of the 2nd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, BigMine '13, pages 95–102, Chicago, Illinois, August 2013. Association for Computing Machinery.

[16] S. Bhadra and A. Ferreira. Complexity of Connected Components in Evolving Graphs and the Computation of Multicast Trees in Dynamic Networks. In Samuel Pierre, Michel Barbeau, and Evangelos Kranakis, editors, *Proceedings of ADHOC-NOW*, number 3 in Lecture Notes in Computer Science, pages 259–270, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[17] Vincent D Blondel, Adeline Decuyper, and Gautier Krings. A survey of results on mobile phone datasets analysis. *EPJ data science*, 4(1):10, 2015.

[18] Benjamin Blonder, Tina W Wey, Anna Dornhaus, Richard James, and Andrew Sih. Temporal dynamics and network analysis. *Methods in Ecology and Evolution*, 3(6):958–972, 2012.

[19] John Adrian Bondy, Uppaluri Siva Ramachandra Murty, et al. *Graph theory with applications*, volume 290. Macmillan London, 1976.

[20] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. Graph structure in the Web. *Computer Networks*, 33(1):309–320, June 2000.

[21] Arnaud Casteigts, Paola Flocchini, Bernard Mans, and Nicola Santoro. Shortest, Fastest, and Foremost Broadcast in Dynamic Networks. *International Journal of Foundations of Computer Science*, 26(04):499–522, June 2015. Publisher: World Scientific Publishing Co.

[22] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-varying graphs and dynamic networks. *International Journal of Parallel, Emergent and Distributed Systems*, 27(5):387–408, 2012.

[23] Arnaud Casteigts, Ralf Klasing, Yessin M. Neggaz, and Joseph G. Peters. Efficiently testing t-interval connectivity in dynamic graphs. In *International Conference on Algorithms and Complexity*, page 89–100, 2015.

[24] Munmun De Choudhury, Hari Sundaram, Ajita John, and Dorée Duncan Seligmann. Social synchrony: Predicting mimicry of user actions in online social media. In *Proc. Int. Conf. on Computational Science and Engineering*, pages 151–158, 2009.

[25] Marco Corneli, Pierre Latouche, and Fabrice Rossi. Block modelling in dynamic networks with non-homogeneous poisson processes and exact icl. *Social Network Analysis and Mining*, 6(1):55, 2016.

[26] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959.

[27] M. De Domenico, A. Lima, P. Mougel, and M. Musolesi. The Anatomy of a Scientific Rumor. *Scientific Reports*, 3(1):1–9, October 2013. Number: 1 Publisher: Nature Publishing Group.

[28] Patrick Doreian and Frans N Stokman. *Evolution of social networks*, volume 1. Psychology Press, 1997.

[29] Stuart E. Dreyfus. An Appraisal of Some Shortest-Path Algorithms. *Operations Research*, 17(3):395–412, June 1969.

[30] P. Erdős and A. Rényi. On the Evolution of Random Graphs. In *PUBLICATION OF THE MATHEMATICAL INSTITUTE OF THE HUNGARIAN ACADEMY OF SCIENCES*, pages 17–61, 1960.

[31] Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, June 1962.

[32] Romain Fontugne, Pierre Borgnat, Patrice Abry, and Kensuke Fukuda. MAWILab: Combining Diverse Anomaly Detectors for Automated Anomaly Labeling and Performance Benchmarking. In *ACM CoNEXT '10*, Philadelphia, PA, December 2010.

[33] Julie Fournet and Alain Barrat. Contact patterns among high school students. *PLoS ONE*, 9(9):e107878, 09 2014.

[34] Zvi Galil and Giuseppe F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys*, 23(3):319–344, September 1991.

[35] Noé Gaumont, Tiphaine Viard, Raphaël Fournier-S'Niehotta, Qinna Wang, and Matthieu Latapy. Analysis of the temporal and structural features of threads in a mailing-list. In *Complex Networks VII*, pages 107–118. Springer, 2016.

[36] Laetitia Gauvin, Mathieu Génois, Márton Karsai, Mikko Kivelä, Taro Takaguchi, Eugenio Valdano, and Christian L. Vestergaard. Randomized reference models for temporal networks. 2020.

[37] Laetitia Gauvin, André Panisson, and Ciro Cattuto. Detecting the community structure and activity patterns of temporal networks: a non-negative tensor factorization approach. *PloS one*, 9(1):e86028, 2014.

[38] Betsy George. *Spatio-temporal networks: modeling and algorithms*. University of Minnesota, 2008.

[39] Luiz H Gomes, Virgilio AF Almeida, Jussara M Almeida, Fernando DO Castro, and Luís MA Bettencourt. Quantifying social and opportunistic behavior in email networks. *Advances in Complex Systems*, 12(01):99–112, 2009.

[40] GroupLens Research. MovieLens data sets. `http://www.grouplens.org/node/73`, October 2006.

[41] László Gulyás, George Kampis, and Richard O Legendi. Elementary models of dynamic networks. *The european physical journal special topics*, 222(6):1311–1333, 2013.

[42] Carlos Gómez-Calzado, Arnaud Casteigts, Alberto Lafuente, and Mikel Larrea. A Connectivity Model for Agreement in Dynamic Systems. In Jesper Larsson Träff, Sascha Hunold, and Francesco Versaci, editors, *Euro-Par 2015: Parallel Processing*, Lecture Notes in Computer Science, pages 333–345, Berlin, Heidelberg, 2015. Springer.

[43] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical Report LA-UR-08-05495; LA-UR-08-5495, Los Alamos National Lab. (LANL), Los Alamos, NM (United States), January 2008.

[44] Ronan Hamon, Pierre Borgnat, Patrick Flandrin, and Céline Robardet. Duality between temporal networks and signals: Extraction of the temporal network structures. *arXiv preprint arXiv:1505.03044*, 2015.

[45] Christopher R Harshaw, Robert A Bridges, Michael D Iannacone, Joel W Reed, and John R Goodall. Graphprints: Towards a graph analytic method for network anomaly detection. In *Proceedings of the 11th Annual Cyber and Information Security Research Conference*, pages 1–4, 2016.

[46] Monika R. Henzinger and Valerie King. Randomized Fully Dynamic Graph Algorithms with Polylogarithmic Time Per Operation. *J. ACM*, 46(4):502–516, July 1999.

[47] Petter Holme and Jari Saramäki. Temporal networks. *Physics Reports*, 519(3):97–125, October 2012.

[48] Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, and Seth Pettie. Fully dynamic connectivity in $O(logn(loglogn)^2)$ amortized expected time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '17, pages 510–520, Barcelona, Spain, January 2017. Society for Industrial and Applied Mathematics.

[49] Charles Huyghues-Despointes, Binh-Minh Bui-Xuan, and Clémence Magnien. Forte delta-connexité dans les flots de liens. In *ALGOTEL 2016 - 18èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications*, Bayonne, France, May 2016.

[50] Lorenzo Isella, Juliette Stehlé, Alain Barrat, Ciro Cattuto, Jean-François Pinton, and Wouter Van den Broeck. What's in a crowd? analysis of face-to-face behavioral networks. *J. of Theoretical Biology*, 271(1):166–180, 2011.

[51] Raj Iyer, David Karger, Hariharan Rahul, and Mikkel Thorup. An Experimental Study of Polylogarithmic, Fully Dynamic, Connectivity Algorithms. *J. Exp. Algorithmics*, 6, December 2001.

[52] Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic Graph Connectivity in Polylogarithmic Worst Case Time. In *Proceedings of the Twenty-fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '13, pages 1131–1142, Philadelphia, PA, USA, 2013. Society for Industrial and Applied Mathematics.

[53] Casper Kejlberg-Rasmussen, Tsvi Kopelowitz, Seth Pettie, and Mikkel Thorup. Faster Worst Case Deterministic Dynamic Connectivity. In Piotr Sankowski and Christos Zaroliagis, editors, *24th Annual European Symposium on Algorithms (ESA 2016)*, volume 57 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 53:1–53:15, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[54] David Kempe, Jon Kleinberg, and Amit Kumar. Connectivity and Inference Problems for Temporal Networks. *Journal of Computer and System Sciences*, 64(4):820–842, June 2002.

[55] Vassilis Kostakos. Temporal graphs. *Physica A: Statistical Mechanics and its Applications*, 388(6):1007–1023, 2009.

[56] Lauri Kovanen, Kimmo Kaski, János Kertész, and Jari Saramäki. Temporal motifs reveal homophily, gender-specific patterns, and group talk in call sequences. *Proceedings of the National Academy of Sciences*, 110(45):18070–18075, 2013.

[57] Gautier Krings, Márton Karsai, Sebastian Bernhardsson, Vincent D Blondel, and Jari Saramäki. Effects of time window size and placement on the structure of an aggregated communication network. *EPJ Data Science*, 1(1):4, 2012.

[58] Renaud Lambiotte and Naoki Masuda. *A guide to temporal networks*, volume 4. World Scientific, 2016.

[59] Matthieu Latapy, Tiphaine Viard, and Clémence Magnien. Stream graphs and link streams for the modeling of interactions over time. *Social Network Analysis and Mining*, 8(1):61, October 2018.

[60] Luigi Laura and Federico Santaroni. Computing Strongly Connected Components in the Streaming Model. In Alberto Marchetti-Spaccamela and Michael Segal, editors, *Theory and Practice of Algorithms in (Computer) Systems*, Lecture Notes in Computer Science, pages 193–205. Springer Berlin Heidelberg, 2011.

[61] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM transactions on Knowledge Discovery from Data (TKDD)*, 1(1):2–es, 2007.

[62] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[63] Ee-Peng Lim, Viet-An Nguyen, Nitin Jindal, Bing Liu, and Hady Wirawan Lauw. Detecting product review spammers using rating behaviors. In *Proc. Int. Conf. on Information and Knowledge Management*, pages 939–948, 2010.

[64] Yannick Léo, Christophe Crespelle, and Eric Fleury. Non-Altering Time Scales for Aggregation of Dynamic Networks into Series of Graphs. *arXiv:1805.06188 [cs]*, May 2018. arXiv: 1805.06188.

[65] Lucie Martinet, Christophe Crespelle, and Eric Fleury. Dynamic contact network analysis in hospital wards. In *Complex Networks V*, pages 241–249. Springer, 2014.

[66] Paolo Massa and Paolo Avesani. Controversial users demand local trust metrics: an experimental study on epinions.com community. In *Proc. American Association for Artificial Intelligence Conf.*, pages 121–126, 2005.

[67] Catherine Matias and Vincent Miele. Statistical clustering of temporal networks through a dynamic stochastic block model. *arXiv preprint arXiv:1506.07464*, 2015.

[68] Othon Michail. An introduction to temporal graphs: An algorithmic perspective. *Internet Mathematics*, 12(4):239–280, 2016.

[69] Alan Mislove. *Online Social Networks: Measurement, Analysis, and Applications to Distributed Information Systems*. PhD thesis, Rice University, 2009.

[70] Paola Modesti and Anna Sciomachen. A utility measure for finding multiobjective shortest paths in urban multimodal transportation networks1This work has been partially supported by the Italian National Research Council (CNR) Project on Transportation "PFT2", subproject 4.2.1, Contract N. 96.00112.PF74.1. *European Journal of Operational Research*, 111(3):495–508, December 1998.

[71] M. E. J. Newman. The Structure and Function of Complex Networks. *SIAM Review*, 45(2):167, 2003.

[72] Mark Newman. *Networks*. Oxford university press, 2018.

[73] V. Nicosia, J. Tang, M. Musolesi, G. Russo, C. Mascolo, and V. Latora. Components in time-varying graphs. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 22(2):023101, June 2012. arXiv: 1106.2134.

[74] Vincenzo Nicosia, John Tang, Cecilia Mascolo, Mirco Musolesi, Giovanni Russo, and Vito Latora. Graph metrics for temporal networks. In *Temporal networks*, pages 15–40. Springer, 2013.

[75] Tore Opsahl and Pietro Panzarasa. Clustering in weighted networks. *Social Networks*, 31(2):155–163, 2009.

[76] Raj Kumar Pan and Jari Saramäki. Path lengths, correlations, and centrality in temporal networks. *Physical Review E*, 84(1):016105, July 2011. Publisher: American Physical Society.

[77] Ashwin Paranjape, Austin R. Benson, and Jure Leskovec. Motifs in Temporal Networks. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, WSDM '17, pages 601–610, Cambridge, United Kingdom, February 2017. Association for Computing Machinery.

[78] Fabiola S.F. Pereira, Sandra de Amo, and João Gama. Evolving Centralities in Temporal Graphs: A Twitter Network Analysis. In *2016 17th IEEE International Conference on Mobile Data Management (MDM)*, volume 2, pages 43–48, June 2016. ISSN: 2375-0324.

[79] Léo Rannou. Straph – Python Library for the modelisation and analysis of Stream Graphs, 2020.

[80] Bruno Ribeiro, Nicola Perra, and Andrea Baronchelli. Quantifying the effect of temporal resolution on time-varying networks. *Scientific reports*, 3:3006, 2013.

[81] Ben Roberts and Dirk P. Kroese. Estimating the Number of s-t Paths in a Graph. *Journal of Graph Algorithms and Applications*, 11(1):195–214, 2007.

[82] Giulio Rossetti, Letizia Milli, Salvatore Rinzivillo, Alina Sîrbu, Dino Pedreschi, and Fosca Giannotti. NDlib : a python library to model and analyze diffusion processes over complex networks. *International Journal of Data Science and Analytics*, 5(1):61–79, February 2018.

[83] Nicola Santoro, Walter Quattrociocchi, Paola Flocchini, Arnaud Casteigts, and Frederic Amblard. Time-varying graphs and social network analysis: Temporal indicators and metrics. *arXiv preprint arXiv:1102.0629*, 2011.

[84] Ingo Scholtes, Nicolas Wider, and Antonios Garas. Higher-Order Aggregate Networks in the Analysis of Temporal Networks: Path structures and centralities. *The European Physical Journal B*, 89(3), March 2016. arXiv: 1508.06467.

[85] Konstantinos Semertzidis, Evaggelia Pitoura, and Kostas Lillis. TimeReach: Historical Reachability Queries on Evolving Graphs, 2015. type: dataset.

[86] Kijung Shin, Tina Eliassi-Rad, and Christos Faloutsos. CoreScope: Graph Mining Using k-Core Analysis — Patterns, Anomalies and Algorithms. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 469–478, December 2016. ISSN: 2374-8486.

[87] Yiannis Siglidis [LIP6. stream-graph: A library for Stream Graphs.

[88] Sandipan Sikdar, Niloy Ganguly, and Animesh Mukherjee. Time series analysis of temporal networks. *The European Physical Journal B*, 89(1):1–11, 2016.

[89] Frédéric Simard. On computing distances and latencies in Link Streams. In *2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 394–397, August 2019. ISSN: 2473-991X.

[90] Ann E Sizemore and Danielle S Bassett. Dynamic graph metrics: Tutorial, toolbox, and tale. *Neuroimage*, 180:417–427, 2018.

[91] Tom AB Snijders, Gerhard G Van de Bunt, and Christian EG Steglich. Introduction to stochastic actor-based models for network dynamics. *Social networks*, 32(1):44–60, 2010.

[92] Christoph Stadtfeld and Per Block. Interactions, actors, and time: Dynamic network actor models for relational events. *Sociological Science*, 4:318–352, 2017.

[93] Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. NetworKit: A tool suite for large-scale complex network analysis, December 2016. DOI: 10.1017/nws.2016.20.

[94] Jimeng Sun, Dacheng Tao, and Christos Faloutsos. Beyond streams and graphs: dynamic tensor analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 374–383, 2006.

[95] Jun Sun, Jérôme Kunegis, and Steffen Staab. Predicting user roles in social networks using transfer learning with feature transformation. In *Proc. ICDM Workshop on Data Mining in Networks*, 2016.

[96] John Tang, Mirco Musolesi, Cecilia Mascolo, and Vito Latora. Temporal distance metrics for social network analysis. In *Proceedings of the 2nd ACM workshop on Online social networks*, WOSN '09, pages 31–36, Barcelona, Spain, August 2009. Association for Computing Machinery.

[97] William Hedley Thompson, Per Brantefors, and Peter Fransson. From static to temporal network theory: Applications to functional brain connectivity. *Network Neuroscience*, 1(2):69–99, 2017.

[98] William Hedley Thompson, Per Brantefors, and Peter Fransson. From static to temporal network theory: Applications to functional brain connectivity. *Network Neuroscience*, 1(2):69–99, April 2017.

[99] Shahadat Uddin, Mahendra Piraveenan, Kon Shing Kenneth Chung, and Liaquat Hossain. Topological analysis of longitudinal networks. In *2013 46th Hawaii International Conference on System Sciences*, pages 3931–3940. IEEE, 2013.

[100] Mathilde Vernet, Yoann Pigne, and Eric Sanlaville. A Study of Connectivity on Dynamic Graphs: Computing Persistent Connected Components. February 2020.

[101] Jordan Viard, Matthieu Latapy, and Clémence Magnien. Revealing contact patterns among high-school students using maximal cliques in link streams.

In *2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 1517–1522. IEEE, 2015.

[102] Tiphaine Viard and Matthieu Latapy. Identifying roles in an ip network with temporal and structural density. In *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 801–806. IEEE, 2014.

[103] Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P. Gummadi. On the evolution of user interaction in facebook. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Social Networks (WOSN'09)*, August 2009.

[104] Stanley Wasserman, Katherine Faust, et al. *Social network analysis: Methods and applications*, volume 8. Cambridge university press, 1994.

[105] Klaus Wehmuth, Artur Ziviani, and Eric Fleury. A unifying model for representing time-varying graphs. In *2015 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 1–10. IEEE, 2015.

[106] Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, NJ, 1996.

[107] Audrey Wilmet, Tiphaine Viard, Matthieu Latapy, and Robin Lamarche-Perrin. Degree-based outliers detection within ip traffic modelled as a link stream. In *2018 Network Traffic Measurement and Analysis Conference (TMA)*, pages 1–8. IEEE, 2018.

[108] H. Wu, Y. Huang, J. Cheng, J. Li, and Y. Ke. Reachability and time-based path queries in temporal graphs. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 145–156, May 2016.

[109] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. Path Problems in Temporal Graphs. *Proc. VLDB Endow.*, 7(9):721–732, May 2014.

[110] Christian Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '13, page 1757–1769, USA, 2013. Society for Industrial and Applied Mathematics.

[111] Kevin S Xu and Alfred O Hero. Dynamic stochastic blockmodels: Statistical models for time-evolving networks. In *International conference on social computing, behavioral-cultural modeling, and prediction*, pages 201–210. Springer, 2013.

[112] B. Bui Xuan, A. Ferreira, and A. Jarry. Computing shortest, fastest, and foremost journeys in dynamic networks. *International Journal of Foundations of Computer Science*, 14(02):267–285, April 2003. Publisher: World Scientific Publishing Co.

[113] Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. DAGGER: A Scalable Index for Reachability Queries in Large Dynamic Graphs. *arXiv:1301.0977 [cs]*, January 2013. arXiv: 1301.0977.

[114] Jeffrey Xu Yu and Jiefeng Cheng. Graph Reachability Queries: A Survey. In Charu C. Aggarwal and Haixun Wang, editors, *Managing and Mining Graph Data*, Advances in Database Systems, pages 181–215. Springer US, Boston, MA, 2010.

[115] Hilmi Yıldırım, Vineet Chaoji, and Mohammed J. Zaki. GRAIL: a scalable index for reachability queries in very large graphs. *The VLDB Journal*, 21(4):509–534, August 2012.

[116] Andy Diwen Zhu, Wenqing Lin, Sibo Wang, and Xiaokui Xiao. Reachability queries on large dynamic graphs: a total order approach. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1323–1334, Snowbird, Utah, USA, June 2014. Association for Computing Machinery.