
Efficient Dynamic Data Structures for Reachability Queries on Large Temporal Graphs

Luiz Fernando Afra Brito



UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Uberlândia
October 17, 2023

Luiz Fernando Afra Brito

**Efficient Dynamic Data Structures for
Reachability Queries on Large Temporal Graphs**

Tese de doutorado apresentada ao Programa de Pós-graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientador: Prof. Dr. Marcelo Keese Albertini

Coorientador: Prof. Dr. Bruno Augusto Nassif Travençolo

Uberlândia
October 17, 2023

Ficha Catalográfica Online do Sistema de Bibliotecas da UFU
com dados informados pelo(a) próprio(a) autor(a).

B862 Brito, Luiz Fernando Afra, 1991-
2023 Efficient dynamic data structures for reachability
queries on large temporal graphs [recurso eletrônico] /
Luiz Fernando Afra Brito. - 2023.

Orientador: Marcelo Keese Albertini.

Coorientador: Bruno Augusto Nassif Travençolo.

Tese (Doutorado) - Universidade Federal de Uberlândia,
Pós-graduação em Ciência da Computação.

Modo de acesso: Internet.

Disponível em: <http://doi.org/10.14393/ufu.te.2023.460>

Inclui bibliografia.

Inclui ilustrações.

1. Computação. I. Albertini, Marcelo Keese ,1984-,
(Orient.). II. Travençolo, Bruno Augusto Nassif,1981- ,
(Coorient.). III. Universidade Federal de Uberlândia.
Pós-graduação em Ciência da Computação. IV. Título.

CDU: 681.3

Bibliotecários responsáveis pela estrutura de acordo com o AACR2:
Gizele Cristine Nunes do Couto - CRB6/2091
Nelson Marcos Ferreira - CRB6/3074

Agradecimentos

Agradeço à minha família e amigos, meus orientadores, prof. Dr. Marcelo Keese Albertini e prof. Dr. Bruno Augusto Nassif Travencolo, ao meu supervisor de estudos na França, prof. Dr. Arnaud Casteigts, que me acompanhou durante minha permanência no *Laboratoire Bordelais de Recherche en Informatique* (LaBRI), ao prof. Dr. Gonzalo Navarro, e aos integrantes da minha banca de defesa, prof. Dr. Felipe Alves da Louza, prof. Dr. Humberto Luiz Razente, prof. Dr. Guilherme Pimentel Telles e prof. Dr. Alan Demétrius Baria Valejo.

Também agradeço à Fundação de Amparo à Pesquisa do Estado de Minas Gerais (FAPEMIG) e à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) por fomentarem o Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Uberlândia e, conseqüentemente, possibilitar o meu estudo em tempo integral.

Resumo

Grafos temporais são ferramentas usadas para representar fenômenos que ocorrem ao longo do tempo. Ao incluir o tempo nos estudos sobre grafos, pode-se descrever sistemas como processos que evoluem continuamente e, assim, descobrir novas soluções em problemas relacionados. Existem várias consultas de alto nível que podem nos ajudar na análise de grafos temporais como a verificação do alcance entre vértices por meio de caminhos temporais e a reconstrução desses caminhos quando possível. Entretanto, tarefas como essas são computacionalmente intensivas e, ademais, nota-se atualmente um aumento substancial do volume de novos dados produzidos. Neste cenário, a hierarquia de memória, incluindo memórias secundárias e caches, deve ser levada em consideração durante o desenvolvimento de aplicações para grafos temporais. Neste trabalho, serão apresentadas estruturas de dados dinâmicas implementadas em memórias primária e secundária que respondem consultas de alcance em grandes grafos temporais. Dentre nossos resultados, destacamos um novo objeto matemático chamado Fecho Transitivo Temporal, uma generalização do Fecho Transitivo comumente estudado em grafos não-temporais. Usando esse novo conceito, propomos três novas estruturas de dados dinâmicas. A primeira estrutura mantém um Fecho Transitivo Temporal em memória primária usando espaço $O(n^2\tau)$, responde consultas de alcance em tempo $O(\log \tau)$ e insere novos contatos em $O(n^2 \log \tau)$, onde τ é a quantidade de etiquetas de tempo e n é o número de vértices em um grafo temporal. A segunda estrutura usa uma representação compacta, também em memória primária, que reduz o espaço utilizado pela estrutura em vários cenários e possui desempenho similar na execução das operações. Finalmente, a terceira estrutura persiste os dados em memória secundária usando espaço $O(n^2\tau)$ e usa algoritmos para responder consultas de alcance e fazer atualizações que acessam, respectivamente, $O(1)$ e $O(n^2\tau/B)$ páginas em disco, onde B é o tamanho de uma página em memória secundária.

Palavras-chave: Grafos temporal, estrutura de dados dinâmica, consulta de alcance entre vértices, fecho transitivo, memória primária, memória secundária.

Abstract

Temporal graphs serve as a modeling tool to represent interactive phenomena that occur over time. By adding the time dimension to these models, we can better describe systems, study them as a continuously evolving process, and discover better solutions to existing problems. There are many high-level queries that aid us in the analysis of temporal graphs, such as checking whether vertices are reachable through temporal paths and reconstructing such temporal paths. However, these tasks are computationally demanding and the increasing volume of data produced at high speeds brings us new challenges. In this scenario, the memory hierarchy, including secondary memory, should be taken into consideration when designing applications for temporal graphs. In this document, we present dynamic data structures for primary and second memories that answer reachability queries on large temporal graphs. During our investigation, we seek strategies that improve the time and space needed to maintain such data structures, and the time to compute reachability queries. Among our results, we highlight a new mathematical object called Timed Transitive Closure (TTC), which generalizes the standard Transitive Closure (TC) concept for temporal graphs. By using this novel object, we propose three new dynamic data structures. The first data structure maintains the TTC information in primary memory using $O(n^2\tau)$ space while answering reachability queries in time $O(\log \tau)$ and inserting new contacts in $O(n^2 \log \tau)$, where τ is the number of timestamps and n is the number of vertices in a temporal graph. The second data structure uses a compact representation, also in primary memory, that greatly reduces the space usage in several scenarios while retaining similar performance. Finally, the third data structure persists the reachability information on disk using $O(n^2\tau)$ space while accessing $O(1)$ pages for answering reachability queries and $O(n^2\tau/B)$ pages for performing updates, where B is the size of a page in secondary memory.

Keywords: Temporal graph, dynamic data structure, reachability queries, transitive closure, primary memory, secondary memory.



UNIVERSIDADE FEDERAL DE UBERLÂNDIA
 Coordenação do Programa de Pós-Graduação em Ciência da Computação
 Av. João Naves de Ávila, 2121, Bloco 1A, Sala 243 - Bairro Santa Mônica, Uberlândia-MG, CEP 38400-902
 Telefone: (34) 3239-4470 - www.ppgco.facom.ufu.br - cpgfacom@ufu.br



ATA DE DEFESA - PÓS-GRADUAÇÃO

Programa de Pós-Graduação em:	Ciência da Computação				
Defesa de:	Tese, 13/2023, PPGCO				
Data:	21 de julho de 2023	Hora de início:	13:39	Hora de encerramento:	17:12
Matrícula do Discente:	11823CCP001				
Nome do Discente:	Luiz Fernando Afra Brito				
Título do Trabalho:	Efficient Dynamic Data Structures for Reachability Queries on Large Temporal Graphs				
Área de concentração:	Ciência da Computação				
Linha de pesquisa:	Ciência de Dados				
Projeto de Pesquisa de vinculação:	-				

Reuniu-se na sala 1B230, Bloco 1B, Campus Santa Mônica, da Universidade Federal de Uberlândia, a Banca Examinadora, designada pelo Colegiado do Programa de Pós-graduação em Ciência da Computação, assim composta: Professores Doutores: Bruno Augusto Nassif Travençolo - FACOM/UFU (Coorientador), Humberto Luiz Razente - FACOM/UFU, Felipe Alves da Louza - FEELT/UFU, Guilherme Pimentel Telles - IC/UNICAMP, Alan Demétrius Baria Valejo - DC/UFSCAR e Marcelo Kesse Albertini - FACOM/UFU, orientador do candidato.

Ressalta-se que o Prof. Dr. Alan Demétrius Baria Valejo participou da defesa por meio de videoconferência desde a cidade de São Carlos - SP. Os outros membros da banca e o aluno participaram *in loco*.

Iniciando os trabalhos a presidente da mesa, Prof. Dr. Marcelo Kesse Albertini, apresentou a Comissão Examinadora e o candidato, agradeceu a presença do público, e concedeu ao Discente a palavra para a exposição do seu trabalho. A duração da apresentação do Discente e o tempo de arguição e resposta foram conforme as normas do Programa.

A seguir o senhor presidente concedeu a palavra, pela ordem sucessivamente, aos examinadores, que passaram a arguir o candidato. Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu o resultado final, considerando o candidato:

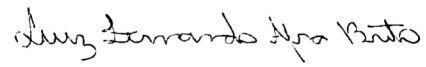
Aprovado

Esta defesa faz parte dos requisitos necessários à obtenção do título de Doutor.

O competente diploma será expedido após cumprimento dos demais requisitos, conforme as normas do Programa, a legislação pertinente e a regulamentação interna da UFU.

Nada mais havendo a tratar foram encerrados os trabalhos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.

I hereby certify that I have obtained all legal permissions from the owner(s) of each third-party copyrighted matter included in my thesis, and that their permissions allow availability such as being deposited in public digital libraries.

A handwritten signature in black ink, reading "Luiz Fernando Afra Brito". The signature is written in a cursive style with some loops and flourishes.

Luiz Fernando Afra Brito

List of Figures

Figure 1 – Perceived time complexity of the Louvain’s algorithm	21
Figure 2 – Illustration of a temporal graph	27
Figure 3 – A temporal graph and its transitive closure	28
Figure 4 – Example of vertex reachability in undirected graphs	38
Figure 5 – Example of vertex reachability in directed graphs	39
Figure 6 – Example of vertex reachability in temporal graphs	42
Figure 7 – Two types of timed transitive closures for a temporal graph	52
Figure 8 – Basic steps to update a timed transitive closure	54
Figure 9 – Number of R-tuples stored in our data structure	60
Figure 10 – A dynamic bit-vector that uses a layout similar to B ⁺ -trees	64
Figure 11 – Representation of a set of non-nested time interval using two bit-vectors	66
Figure 12 – Sequence of insertions using our data structure based on bit-vectors . .	68
Figure 13 – Comparison of data structures to represent a set of non-nested intervals	74
Figure 14 – Comparison of TTCs using data structures to represent sets of non- nested intervals for each pair of vertices	75
Figure 15 – Temporal graph and its associated disk-based timed temporal closure .	82
Figure 16 – Maintenance of our disk-based timed transitive closure	85
Figure 17 – Time to maintain our data structures on synthetic data	88
Figure 18 – Illustration of the process performed by our update algorithm	90
Figure 19 – Time-interval Log per Edge representation	111
Figure 20 – Adjacency Log of Events representation	113
Figure 21 – Compact Adjacency Sequence representation	115
Figure 22 – Compressed Events ordered by Time representation	118
Figure 23 – Temporal Graph Compressed Suffix Array representation	120
Figure 24 – k^d tree representation, with $k = 2$ and $d = 2$, of a temporal graph . . .	123

List of Tables

Table 1 – Examples of queries applied to a temporal graph 29

Table 2 – Space for storing temporal graphs and timed transitive closures 78

Table 3 – Wall-clock time to insert shuffled contacts from real-world datasets 91

Table 4 – Worst-case space cost of temporal graph representations 125

Table 5 – Cost of `has_edge` and `neighbors` queries for different representations 126

Table 6 – Cost of `neighborsr` and `aggregate` queries for different representations 126

List of Algorithms

1	<code>add_contact(u, v, t)</code>	57
2	<code>reconstruct_journey(u, v, t_1, t_2)</code>	57
3	<code>FIND_PREV($(D, A), t$)</code>	67
4	<code>FIND_NEXT($(D, A), t$)</code>	67
5	<code>INSERT($(D, A), t_1, t_2$)</code>	69
6	<code>JOIN(N_1, N_2)</code>	71
7	<code>SPLIT_AT_JTH_ONE(N, j)</code>	72
8	<code>add_contact(u, v, t)</code>	86
9	<code>reconstruct_journey(u, v, t_1, t_2)</code>	87
10	<code>join(T_{left}, T_{right})</code>	132
11	<code>split(T, L)</code>	133

Acronyms list

BST Binary Search Tree

BFS Breath-First Search

CAS Compact Adjacency Sequence

CET Compressed Events ordered by Time

CSA Compressed Suffix Array

DAG Directed Acyclic Graph

DFS Depth-First Search

EdgeLog time-interval Log per Edge

EveLog adjacency Log of Events

ETDC End-Tagged Dense Codes

HDD Hard Disk Drive

HAMR Heat-Assisted Magnetic Recording

HDMR Heated-Dot Magnetic Recording

IoT Internet of Things

I/O Input/Output

LCA Least Common Ancestor

LRU Least Recently Used

MAMR Microwave-Assisted Magnetic Recording

MST Minimum Spanning Tree

SA Suffix Array

SCC Strongly Connected Component

SMR Shingled Magnetic Recording

TGCSA Temporal Graph Compressed Suffix Array

TC Transitive Closure

TTC Timed Transitive Closure

Contents

1	INTRODUCTION	17
1.1	Motivation	19
1.2	Opportunities and Challenges	22
1.3	Hypothesis and Goals	23
1.4	Contributions	24
1.5	Organization	25
2	FUNDAMENTALS	26
2.1	Temporal Graphs	26
2.2	Queries on Temporal Graphs	28
2.2.1	Low-level Queries for Temporal Graphs	28
2.2.2	High-level Queries for Temporal Graphs	30
2.3	External Memory and Dynamic Data Structures	31
2.3.1	External Data Structures	32
2.4	Concluding remarks	36
3	RELATED WORK	37
3.1	Reachability Queries	37
3.1.1	Reachability on Undirected Graphs	37
3.1.2	Reachability on Directed Graphs	38
3.1.3	Reachability on Temporal Graphs	41
3.2	Reachability Queries on Disk	43

3.2.1	Reachability on Undirected Graphs Stored on Disk	43
3.2.2	Reachability on Directed Graphs Stored on Disk	43
3.2.3	Reachability on Temporal Graphs Stored on Disk	44
3.3	Space-Efficient Data Structures for Querying Temporal Graphs in Primary Memory	45
3.4	Concluding remarks	45
4	A DYNAMIC DATA STRUCTURE FOR TEMPO- RAL REACHABILITY WITH UNSORTED CONTACT INSERTIONS	47
4.1	Reachability Tuples and Timed Transitive Closure	48
4.1.1	Reachability Tuples (R-tuples)	49
4.1.2	Timed Transitive Closure	51
4.2	The Four Operations	55
4.2.1	Reachability and Connectivity Queries	55
4.2.2	Update Operation	55
4.2.3	Journey Reconstruction	56
4.2.4	Evolution of the Number of R-tuples over the Insertions	59
4.3	Concluding remarks	61
5	A COMPACT DATA STRUCTURE FOR TEMPORAL REACHABILITY	62
5.1	Dynamic bit-vectors	63
5.2	Dynamic compact data structure for temporal reachability	65
5.2.1	Compact representation of non-nested intervals	66
5.2.2	Query algorithms	67
5.2.3	New dynamic bit-vector operation to improve interval insertion	69
5.3	Experiments	73
5.3.1	Comparison of data structures for sets of non-nested intervals	73
5.3.2	Comparison of data structures for Time Transitive Closures	75
5.4	Concluding remarks	76

6	A DISK-BASED DATA STRUCTURE FOR TEMPORAL REACHABILITY	77
6.1	Disk-Based Timed Transitive Closure	80
6.1.1	Expanded Reachability Tuples (Expanded R-tuples)	80
6.1.2	Encoding the TTC on Disk	81
6.2	The Four Operations	83
6.2.1	Reachability and Connectivity Queries	83
6.2.2	Update Operation	83
6.2.3	Journey Reconstruction	86
6.3	Experiments	87
6.3.1	Experiments with Synthetic Data	88
6.3.2	Experiments with Real-World Datasets	89
6.4	Concluding remarks	91
7	CONCLUSION	93
7.1	List of publications	95
	BIBLIOGRAPHY	97

APPENDIX **109**

APPENDIX A	– SPACE-EFFICIENT DATA STRUCTURES FOR QUERYING TEMPORAL GRAPHS IN PRIMARY MEMORY	110
A.1	Time-Interval Log per Edge	111
A.1.1	Operation <code>has_edge</code>	112
A.1.2	Operation <code>neighbors</code>	112
A.1.3	Operation <code>neighbors^r</code>	112
A.2	Adjacency Log of Events	112
A.2.1	Operation <code>has_edge</code>	113
A.2.2	Operation <code>neighbors</code>	114
A.2.3	Operation <code>neighbors^r</code>	114

A.3	Compact Adjacency Sequence	114
A.3.1	Operation <code>has_edge</code>	116
A.3.2	Operation <code>neighbors</code>	117
A.3.3	Operation <code>neighbors^r</code>	117
A.4	Compressed Events Ordered by Time	118
A.4.1	Operation <code>has_edge</code>	118
A.4.2	Operation <code>neighbors</code>	119
A.4.3	Operation <code>neighbors^r</code>	119
A.5	Temporal Graph Compressed Suffix Array	119
A.5.1	Operation <code>has_edge</code>	121
A.5.2	Operation <code>neighbors</code>	121
A.5.3	Operation <code>neighbors^r</code>	121
A.6	Compressed k^d Tree	122
A.6.1	Operation <code>has_edge</code>	124
A.6.2	Operation <code>neighbors</code>	124
A.6.3	Operation <code>neighbors^r</code>	124
A.7	Considerations	125
APPENDIX B	– JOIN AND SPLIT OPERATIONS FOR	
	B⁺-TREES	130
B.1	Join operation for B⁺-trees	131
B.2	Split operation for B⁺-trees	132

Introduction

Temporal graphs serve as a modeling tool to represent interactive phenomena that occur over time (MICHAIL, 2016). They describe complex systems as relationships among entities that often appear as contacts or events defining when relationships begin or end. Studies have applied temporal graphs to abstract problems such as: the continuous communication among participants of social networks, in order to detect hierarchies (YANG et al., 2011); the evolution of communities, in order to understand and predict future events (LIBEN-NOWELL; KLEINBERG, 2007); and the flow in transportation networks, in order to check the existence of trajectories and reconstruct them (WU et al., 2017; WILLIAMS; MUSOLESI, 2016; BEDOGNI; FIORE; GLACET, 2018). By adding the time dimension to these models, we can describe such systems, study them as a continuously evolving process, and discover better solutions to existing problems.

There are many high-level queries that aid us in the analysis of temporal graphs. A very important one, which we are particularly interested, is to check whether any two entities can reach each other by traversing contacts through time. These time-respecting paths are known as journeys, and reconstructing them can be very useful as well. For instance, during scenarios of epidemics, information containing the interaction details among infected and non-infected individuals is registered incrementally in a database. Then, this information is periodically queried in order to better understand the dissemination process and, thus, to support actions that slow it down or completely interrupt it (XIAO et al., 2018; XIAO; ASLAY; GIONIS, 2018; ENRIGHT et al., 2021; ROZEN-SHTEIN et al., 2016). However, the collected data can arrive outdated, even though, sometimes, it is important to include this information for future analysis. Think when an infected patient goes to a hospital because he is feeling sick and the healthcare professional discovers that his infection occurred three days ago.

The detection of evolving communities is another important high-level query and knowing the reachability information in advance is fundamental to implement algorithms for this purpose. The main idea is to track the evolution of entity clusters, in which

members in the same cluster are more likely to relate than members in different clusters. By grouping related entities together, we can uncover the topological structure of temporal graphs over time and use higher abstractions to better understand the system. For example, in visualization, one of the most promising areas (SAHU et al., 2017), there are studies that draw strongly connected entities together in order to minimize clutter and improve readability of the whole temporal graph (LINHARES et al., 2017).

However, such tasks can be computationally demanding since algorithms need to process every contact, considering the entire lifetime of the temporal graph. Moreover, recent technological advances such as the Internet of Things (IoT) and the integration of social media concepts in diverse applications have also enabled new content to be created by anyone (ATZORI et al., 2012). This increasing volume of data produced at high speeds brings us new challenges.

On one hand, we need efficient computational mechanisms to persist data that evolve continuously in cheap external storage. On the other hand, we need specialized techniques to load these data in faster (and more expensive) memories using minimal space and process queries as fast as possible in order to extract valuable knowledge. Therefore, data structures for this context must consider all the memory hierarchy, including networking when necessary, and be dynamic. Here, the adjective dynamic refers to the fact that the data structure can be updated after the input data is changed.

There are three types of dynamic data structures: incremental, decremental, and fully-dynamic. Incremental and decremental data structures support only inserting or deleting elements, respectively, while fully-dynamic data structures support both operations. For temporal graphs, an important aspect of data structures is whether the order of updates respects the order of the contacts themselves. Here, data structures can be chronological, whether they operate on a sorted sequence of contacts, or non-chronological, whether they operate on an unsorted sequence of contacts. For instance, during scenarios of epidemics, outdated information containing the interaction details among individuals is reported in an arbitrary order. Thus, a data structure for this scenario should be, at least, incremental and non-chronological.

In this thesis, we consider the problem of maintaining reachability information of a temporal graph \mathcal{G} through an incremental and non-chronological data structure in order to answer reachability queries. In the following, we present the four operations such data structure must support, where, by convention, u and v are entities in \mathcal{G} , and t , t_1 , and t_2 are timestamps:

- `add_contact(u, v, t)`: Updates information based on a contact from u to v at time t . This operation may be called every time a new contact is discovered in the considered scenario.

- `can_reach(u, v, t_1, t_2)`: Returns true if u can reach v through a journey within the interval $[t_1, t_2]$.
- `is_connected(t_1, t_2)`: Returns true if \mathcal{G} , restricted to the interval $[t_1, t_2]$, is temporally connected, *i.e.*, all entities can reach each other through a journey within the interval $[t_1, t_2]$. This operation allows one to test, for example, whether all the nodes of a communication network can reach each other.
- `reconstruct_journey(u, v, t_1, t_2)`: Returns a journey (if one exists) from u to v occurring within the interval $[t_1, t_2]$. This operation may be called, for example, to learn a potential contamination chain from a person to another in a scenario of epidemics.

1.1 Motivation

A naïve approach for our problem stores and updates the temporal graph itself as a set of contacts, then it runs standard journey computation algorithms (XUAN; FERREIRA; JARRY, 2003a) for every new query. However, this approach is adapted only for scenarios in which the number of insertions is much larger than the number of queries, since every query may traverse the whole temporal graph. In contrast, dynamic structures offer a tradeoff between query time, update time, and space usage.

To the best of our knowledge, the only existing work supporting non-chronological contact insertions and exploiting intermediate representations for speeding up reachability queries in temporal graphs is (WU et al., 2016). Their solution relies on maintaining a Directed Acyclic Graph (DAG) in which every original vertex is copied up to τ times (where τ is the number of timestamps) and a journey exists from u to v in the interval $[t, t']$ if and only if vertex u_t can reach vertex $v_{t'}$ in the DAG. However, even though their experiments show their algorithms are efficient on the average case, the worst-case complexities of their algorithms do not improve over the naïve approach. The worst-case query time corresponds to a standard path search (*e.g.*, depth-first search) in the DAG, which takes $\Theta(n^2\tau)$ time with dense temporal graphs, where n is the number of entities. The worst-case update time also corresponds to a standard path search in order to update the affected vertex labels in the DAG. Furthermore, the space complexity (size of the DAG) corresponds essentially to the number of contacts, thus $\Theta(n^2\tau)$ in the worst case. Finally, their data structure only works in primary memory, which is not suitable for working with large temporal graphs. Large in this context means that temporal graphs do not fit entirely in primary memory and, therefore, we need to store them in slower storages.

Storing and querying data structures for large temporal graphs may be difficult, especially when the amount of data grows unbounded (CARO; RODRÍGUEZ; BRISABOA,

2015). Consider the scenario in which one million people use bluetooth devices that register when and who gets close to each other and send this information to a centralized server. Consider also that each individual makes in average 30 contacts per day. In this setting, the server would require at least 100 GB of space in less than a year to store just the plain contacts. If one needs to support fast reachability queries (*e.g.*, can a piece of information created by an individual arrive at another individual’s device by replicating itself to other devices in contact?), it would be necessary even more space to maintain an additional data structure.

In such scenarios, we need to use specialized data structures and algorithms to manipulate large temporal graphs efficiently in both primary memory (CARO; RODRÍGUEZ; BRISABOA, 2015; CARO et al., 2016; BRISABOA et al., 2018; HAN et al., 2014; LABOUSEUR et al., 2015; KHURANA; DESHPANDE, 2013) and secondary memory (BUCHSBAUM et al., 2000; QIAO, 2013; ZHANG et al., 2012a; HIRVISALO; NUUTILA; SOISALON-SOININEN, 1996; CHENG; YU; TANG, 2006; SHIRANI-MEHR; KASHANI; SHAHABI, 2012; STRZHELETSKA, 2018) due to the high cost of managing data only in primary memory and the high risk of losing information in volatile storages. Data structures for secondary memory consider high latency of access and low transfer rate of data (HAN et al., 2014; NEUMANN; WEIKUM, 2010; LABOUSEUR et al., 2015; KHURANA; DESHPANDE, 2013). A strategy to improve operations in secondary memory should reduce the number of disk accesses by maintaining coherent subsets of data near each other (LABOUSEUR et al., 2015). Thus, algorithms can read entire blocks from secondary memory and take advantage of pre-fetched data, mitigating the high latency problem. Data structures for primary memory, differently, consider better latency and transfer rate, whereas they account for less space availability (CARO; RODRÍGUEZ; BRISABOA, 2015; CARO et al., 2016; BRISABOA et al., 2018). In this case, strategies can spend more instructions per algorithm, since the cost of accessing data in primary memory is much lower than accessing data in secondary memory. However, they do not work with large datasets because of their limited space.

Even when using data structures for both primary and secondary memory, high-level queries on temporal graphs can still degrade severely if the data structures do not integrate well. To exemplify this behavior, we conducted a small experiment to compare the wall clock time of the Louvain’s algorithm (BLONDEL et al., 2008) — an algorithm for community detection based on modularity optimization. For this experiment, we implemented a disk-based temporal adjacency list data structure (XUAN; FERREIRA; JARRY, 2003b) using B^+ -trees — a dynamic data structure for secondary memory based on m -ary trees — and a cache data structure in primary memory. Our cache data structure uses the Least Recently Used (LRU) policy (JOHNSON; SHASHA, 1994) and stores nodes of B^+ -trees as blocks. To test these data structures we generated temporal graphs using the Barabási-Albert model (ALBERT; BARABÁSI, 2002) and executed the Louvain’s

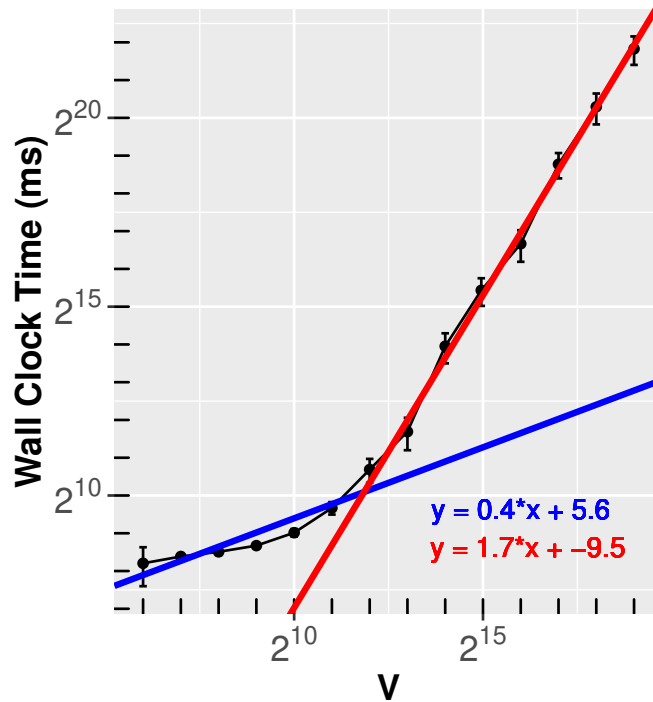


Figure 1 – Perceived time complexity of the Louvain’s algorithm (BLONDEL et al., 2008). This figure shows the results in a log-log plot where the x axis is the number of vertices in the temporal graph and the y axis is the wall clock time to execute the algorithm. We also drew a red and a blue line representing linear regressions, along with their equation, representing the growth of wall clock time in periods where temporal graphs fit entirely in cache and when it does not. We note that when $V \geq 2^{13}$ the curve inclination changes considerably. At this point, the temporal graph does not fit entirely in cache and the algorithm access the secondary memory more often.

algorithm varying the number of vertices at every timestamp.

Figure 1 shows the results in a log-log plot where the x axis is the number of vertices in the temporal graph and the y axis is the wall-clock time to compute the algorithm. The inclination coefficient of fitted straight lines gives us empirically, the overall time complexity. We see that, when the temporal graph can fit entirely in primary memory, the wall-clock time is sublinear by looking at the inclination of the first line. However, as soon as the temporal graph cannot fit entirely in primary memory, and the number of disk access increases, the wall-clock time grows rapidly, becoming subquadratic. For each snapshot at timestamp t , the Louvain’s algorithm needs to traverse the temporal graph many times using a circular access pattern and the LRU cache in memory cannot improve its computation since most hit attempts fail. Therefore, we need to consider more suitable data structures in both primary and secondary memory in order to prevent this degradation.

1.2 Opportunities and Challenges

There are many studies on dynamic data structures for reachability queries on standard (non-temporal) graphs. Usually, they maintain the Transitive Closure (TC) of an input graph (ITALIANO, 1986; ŁĄCKI, 2013; KING; SAGERT, 2002; RODITTY, 2008), which answer reachability queries as fast as possible, but updates are costly; or a vertex-labeling schema (SEUFERT et al., 2013; WEI et al., 2018; YILDIRIM; CHAOJI; ZAKI, 2012; CHEN; GUPTA; KURUL, 2005; VELOSO et al., 2014), which has different tradeoffs. Yet, other studies focus primarily on disk-based data structures in order to maintain information related to large temporal graphs (HIRVISALO; NUUTILA; SOISALON-SOININEN, 1996; AGRAWAL; BORGIDA; JAGADISH, 1989; ZHANG et al., 2012b; ZHANG et al., 2018).

However, there are only few studies on such data structures for temporal graphs. In fact, the area of temporal graphs is relatively new compared to standard graphs. Thus, many problems are still open. For example, does the approaches used to solve reachability on standard graphs can be applied to temporal graphs or does the temporal nature of the temporal graphs demand different approaches? Can data structures for temporal graphs be as fast as the data structures for standard graphs?

In this work, we studied the maintenance of TCs for temporal graphs in order to answer reachability queries. Briefly, the standard TC of a temporal graph is a matrix such that if vertex u reaches v through a journey, then $TC(u, v) = true$. However, this notion is not sufficient to maintain the TC itself dynamically, in the context of temporal graphs, because it does not allow one to decide if a new contact can be composed with previously known journeys. For instance, if we add a contact (u, w, t) , then a contact $(w, v, t - 1)$, we know that we cannot compose a journey from u to v since the first contact happens after the second one; however, with a standard TC, we cannot know if it is possible or not to compose such a journey because it does not keep temporal information.

The maintenance of TCs for large temporal graphs can be quite costly in terms of space usage, especially if we want to include the temporal aspect into standard TCs. Therefore, we need to consider using efficient data structures in secondary memory, which try to reduce the number of Input/Output (I/O) accesses, and exploit data locality in order to keep related data contiguous. Compression techniques can also reduce space and, as a result, reduce the number of I/O accesses as more data can fit into a single block. In (ROY; MIHAJLOVIC; ZWAENEPOEL, 2013), the authors use some of these ideas to improve the processing of large graphs. However, their strategy only works for standard graphs and modifications are not easily implemented. So, we investigated an alternative approach for temporal graphs.

There is a growing area of study on compact data structures that uses only $O(Z)$

space, where Z is the optimal number of bits needed to store some data, targeting mainly primary memory (GAGIE; NAVARRO; PUGLISI, 2012; GROSSI; VITTER, 2005; CHAMBI et al., 2016). There are also two other categories of efficient data structures, the succinct and the implicit data structures, which use $Z + o(Z)$ and $Z + O(1)$ space, respectively (JACOBSON, 1988). These data structures consider space lower bounds based on Information Theory concepts (NAVARRO, 2016), and provide useful queries with little or no costly preprocessing steps such as decompression (GROSSI; GUPTA; VITTER, 2003). Recently, some authors have used compact data structures to store and query temporal graphs in primary memory (CARO; RODRÍGUEZ; BRISABOA, 2015; CARO et al., 2016; BRISABOA et al., 2018). For example, Brisaboa et al. (2018) used a compact version of the suffix array to store text representations of temporal graphs and offer basic queries, such as checking if a relation is active at some timestamp, in logarithmic time at the average cost of 50 to 90 bits per contact $(u, v, t_{begin}, t_{end})$, as empirical experiments suggest, where u and v are entities that relates from time t_{begin} to time t_{end} .

Compact data structures have great potential to be used as a cache for temporal graphs stored in secondary memory. However, although compact data structures can reduce space usage, most of them are static and do not allow modifications. When compact data structures support update operations, their operations have at least an additional $O(\log n)$ factor in their costs, and the space usage can be worse than their static versions (NAVARRO, 2016). These results are under the RUM conjecture (ATHANASOULIS et al., 2016), which states that a data structure cannot optimize simultaneously the time to query information, the time to update itself, and the total space used to organize data.

1.3 Hypothesis and Goals

Our hypothesis for this thesis is:

Hypothesis. *Using specialized data structures for both primary and secondary memory can improve the maintenance of reachability queries on large temporal graphs.*

Our primary goal to verify this hypothesis was to *propose dynamic data structures to handle large temporal graphs that can evolve and cannot be entirely in primary memory*. As presented earlier, even though the algorithms for temporal graphs do not change, their overall time complexity can increase considerably when primary memory is not enough and algorithms start accessing secondary memory. In this thesis, we developed efficient data structures to maintain and query temporal graphs in secondary memory. Therefore, these data structures aim to reduce the amount of I/O accesses in order to mitigate the degradation due to the low cache hit rate and, as a result, high workload in secondary memory during the computation of reachability queries.

In order to reach our primary goal, we studied the following specific goals. The first specific goal was to *develop a base model and implement a simple data structure for primary memory capable of answering reachability queries on temporal graphs*. As a first effort, we studied a new mathematical object based on the standard TC, which also includes the time aspect of journeys, and then studied a preliminary data structure that maintains this object in primary memory. A data structure for reachability query should provide better performance than just performing journey computation algorithms on the temporal graph itself with no special organization, such as those in (XUAN; FERREIRA; JARRY, 2003a). It should also provide better worst-case costs for adding new contacts and answer reachability queries than those introduced in (WU et al., 2016).

The second specific goal was to *implement a space efficient data structure for primary memory capable of efficiently answering reachability queries on temporal graphs*. By improving the space efficiency of data structures in primary memory, we increase the amount of information we can use in faster memories. Consequently, we improve the overall performance of maintaining reachability information for larger temporal graphs since accesses to slower storages is reduced. In order to improve space efficiency, we studied a dynamic compact data structure to incrementally maintain reachability information. We note that most compact data structures for temporal graphs in the literature are static, which means that the data structure is fully rebuilt for any data modification.

Finally, the third specific goal was to *implement a data structure for secondary memory capable of efficiently answering reachability queries on large temporal graphs*. As we are dealing with scenarios where the incoming data does not fit entirely in primary memory, the temporal graph data should be persisted in secondary memory, and the techniques used should be optimized for this type of storage. For example, data structures in secondary memory must take advantage of data locality and minimize accesses to pages in order to improve performance. Therefore, we studied a novel disk-based data structure, considering these characteristics, based on our previous data structure developed for primary memory.

1.4 Contributions

In this thesis, we contributed with the following. A new mathematical object called Timed Transitive Closure (TTC), which generalizes the concept of standard Transitive Closures (TCs) for temporal graphs. A in-memory dynamic data structure that encodes the reachability information of a temporal graph using $O(n^2\tau)$ space, which is essentially the space complexity to store the contacts of a temporal graph; while answering `add_contact(u, v, t)` in time $O(n^2 \log \tau)$, `can_reach(u, v, t_1, t_2)` in $O(\log \tau)$, `is_connected(t_1, t_2)` in $O(n^2 \log \tau)$, and `reconstruct_journey(u, v, t_1, t_2)`

in $O(k \log \tau)$, where n and τ are, respectively, the number entities and the number of timestamps in the lifespan of a temporal graph, and k is the length of the reconstructed journey. A compact dynamic data structure, also for primary memory, that retains the same time complexities for all operations while spending much less space in several scenarios. Finally, a disk-based dynamic data structure that improves performance on most datasets considered in our experiments while accessing $O(n^2\tau/B)$ sequential pages to perform `add_contact`(u, v, t), $O(1)$ pages for `can_reach`(u, v, t_1, t_2), $O(n^2/B)$ pages for `is_connected`(t_1, t_2), and $O(n/B)$ for `reconstruct_journey`(u, v, t_1, t_2), where B is the size of a disk page.

1.5 Organization

The rest of this document is organized as follows. In Chapter 2, we introduce background concepts about temporal graphs, including definition, general strategies for data maintenance and basic queries; and about secondary memory, including basic data structures. In Chapter 3, we review specialized data structures (in both primary and secondary memory) for answering reachability queries on temporal graphs. In Chapter 4, we present our first contribution that introduces a new concept called Timed Transitive Closure (TTC) in order to check reachability of entities and reconstruct temporal paths. In Chapter 5, we present our second contribution that proposes a dynamic compact data structure that improves space efficiency in primary memory. In Chapter 6, we present our third contribution that develops an efficient implementation of our TTC in secondary memory. Finally, in Chapter 7 we draw our conclusions.

Fundamentals

In this chapter, we introduce the background concepts used throughout this document. In Section 2.1, we define temporal graphs and general concepts, such as reachability and Transitive Closure (TC). In Section 2.2, we present basic queries commonly used when analyzing temporal graphs categorizing them into low-level and high-level queries. Finally, in Section 2.3, we introduce important concepts related to disk storage and present basic data structures for working with data in secondary memory.

2.1 Temporal Graphs

Temporal graphs model relationships among entities over time by also describing the evolution of networks.

Definition 1 (Temporal graph as labeled edges). *Following the formalism in (CASTEIGTS et al., 2011), a temporal graph is a tuple $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$, where V is a set of vertices, $E \subseteq V \times V$ is a set of edges, \mathcal{T} is the time interval over which the network exists (lifetime), $\rho : E \times \mathcal{T} \rightarrow \{0, 1\}$ is a presence function that expresses whether an edge is present at a time instant, and $\zeta : E \times \mathcal{T} \mapsto \mathbb{T}$ is a latency function that expresses the duration to occur an interaction for an edge at a timestamp, where \mathbb{T} is the time domain (typically \mathbb{R} or \mathbb{N}).*

Temporal graphs can be directed or undirected. Directed edges express uni-directional edges as in followers-followees networks, such that pairs $(u, v) \in E$ are ordered, meaning that $(u, v) \neq (v, u)$. Differently, undirected edges express bi-directionality as in collaboration networks in which authors co-write papers, such that the pairs $(u, v) \in E$ are unordered and $(u, v) = (v, u)$. Figure 2 illustrates a directed temporal graph.

In this work, we consider a setting where E is a set of directed edges, \mathbb{T} is equal to \mathbb{N} (time is discrete) and $\mathcal{T} = [1, \tau]$ (the lifetime contains τ timestamps). The latency function is constant, namely $\zeta = \delta$, where δ is any fixed positive integer. We call (u, v, t)

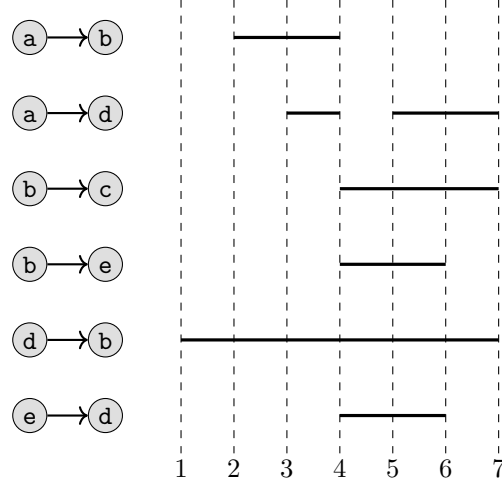


Figure 2 – Temporal graph $\mathcal{G} = (V, E, T, \rho, \zeta)$ that has the set of vertices $V = \{a, b, c, d, e\}$, the set of edges $E = \{(a, b), (a, d), (b, c), (b, e), (d, b), (e, d)\}$, the lifetime $\mathcal{T} = [1, \tau]$, where $\tau = 7$, a presence function ρ describing the set of persistent contacts $C = \{(a, b, 2, 4), (a, d, 3, 4), (a, d, 5, 7), (b, c, 4, 7), (b, e, 4, 6), (d, b, 1, 7), (e, d, 4, 6)\}$, and a latency function $\zeta = \delta$, where $\delta = 0$, *i.e.*, interactions between vertices are instantaneous.

a *contact* in \mathcal{G} if $\rho((u, v), t) = 1$ and (u, v, t_1, t_2) a *persistent contact* in \mathcal{G} if $\rho((u, v), t') = 1$ for $t_1 \leq t' \leq t_2$.

We define reachability in temporal graphs in a time-respecting way, by requiring that a path travels along non-decreasing ($\delta = 0$) or increasing ($\delta \geq 1$) timestamps. These paths are called temporal paths or journeys interchangeably.

Definition 2 (Journey). *A journey from u to v in \mathcal{G} is a sequence of contacts $\mathcal{J} = \langle c_1, c_2, \dots, c_k \rangle$, whose sequence of underlying edges form a valid time-respecting path from u to v . For each contact $c_i = (u_i, v_i, t_i)$, it holds that $\rho((u_i, v_i), t_i) = 1$, $v_i = u_{i+1}$, and $t_{i+1} \geq t_i + \delta$ for $i \in [1, k - 1]$. We say that $\text{departure}(\mathcal{J}) = t_1$, $\text{arrival}(\mathcal{J}) = t_k + \delta$ and $\text{duration}(\mathcal{J}) = \text{arrival}(\mathcal{J}) - \text{departure}(\mathcal{J})$. A journey is trivial if it comprises a single contact.*

Definition 3 (Reachability). *A vertex u can reach a vertex v iff there is a journey \mathcal{J} from u to v in \mathcal{G} .*

The standard Transitive Closure (TC) for a temporal graph \mathcal{G} is a directed graph $\mathcal{G}^* = (V, E^*)$ such that $(u, v) \in E^*$ if and only if u can reach v in \mathcal{G} . Figure 3 illustrates a TC. In (BARJON et al., 2014), the authors incrementally maintain \mathcal{G}^* for contacts discovered in a chronological order. However, for contacts discovered in a non-chronological order, the present information of \mathcal{G}^* is not sufficient to decide whether a new contact can be composed with previously known journeys, which motivates the definition of more powerful objects.

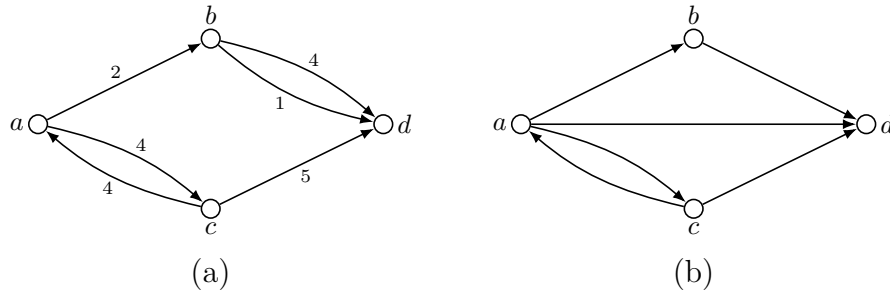


Figure 3 – A temporal graph and its transitive closure. (Left) A temporal graph \mathcal{G} on four vertices $V = \{a, b, c, d\}$, where the presence times of edges are depicted by labels. Whether $\delta = 0$ or $\delta = 1$, this graph has only two non-trivial journeys, namely $\mathcal{J}_1 = \langle (a, b, 2), (b, d, 4) \rangle$ and $\mathcal{J}_2 = \langle (a, c, 4), (c, d, 5) \rangle$. (Right) TC \mathcal{G}^* . Note that \mathcal{J}_1 and \mathcal{J}_2 are represented by the same edge in \mathcal{G}^* (the two contacts from b to d as well).

2.2 Queries on Temporal Graphs

We can perform several queries on temporal graphs in order to analyze their properties. There are two types of queries, according to Bernardo et al. (2013): high-level and low-level queries. Low-level queries solve low abstraction tasks, such as checking if an edge is active at time t or retrieving all neighbors of vertex u at time t . High-level queries solve high abstraction problems such as finding a journey connecting two vertices through time or clustering vertices along snapshots to detect community evolution.

2.2.1 Low-level Queries for Temporal Graphs

Bernardo et al. (2013) categorized low-level queries into three types: edge, vertex, and time-related queries. In all these types, we pass as input time-related arguments. In the case we want to make a query regarding a specific timestamp, we call it a *point-time query*. Instead, if we want to make a query on a range or interval, we call it an *interval query*. Interval queries can also have two different semantics, *weak* or *strong*. If an interval query has weak semantics, it is enough that a condition holds for snapshots of any moment during the interval. Otherwise, if an interval query has strong semantics, some condition needs to hold for all intervals. In this text, if an interval $[t_1, t_2]$ has only one value, when $t_1 = t_2$, we consider it a point-time query.

Edge-related queries retrieve edge information at a time t or in an interval $[t_1, t_2]$. For example, `has_edge(u, v, t)` checks if there is an edge $(u, v) \in E$ active at time t . In order to answer this query, we simply check if there is a contact (u, v, t) . Similarly, `has_edge(u, v, t_1, t_2)` also checks the existence of an edge $(u, v) \in E$, however, now we want to know if this edge is active during an interval. If this query has weak semantics, it is enough to exist a contact (u, v, t') that satisfies $t_1 \leq t' \leq t_2$. Instead, if this query has strong semantics, we check if there is a persistent contact (u, v, t'_1, t'_2) such that $t'_1 \leq$

Table 1 – Examples of queries applied to the temporal graph shown in Figure 2

Query	$t_1 = 3, t_2 = 3$	$t_1 = 3, t_2 = 5$		
	point-time query	weak interval query	strong interval query	
edge	<code>has_edge(a, b, t_1, t_2)</code>	<i>true</i>	<i>true</i>	<i>false</i>
	<code>next_activation(b, c, t_1, t_2)</code>	4	-	-
vertex	<code>neighbors(d, t_1, t_2)</code>	{ <i>b</i> }	{ <i>b</i> }	{ <i>b</i> }
	<code>neighbors^r(d, t_1, t_2)</code>	{ <i>a</i> }	{ <i>a, e</i> }	{}
time	<code>aggregate(t_1, t_2)</code>	{(<i>a, b</i>), (<i>a, d</i>), (<i>d, b</i>)}	{}	{}
	<code>activated_edges(t_1, t_2)</code>	{(<i>a, d</i>)}	{}	{}
	<code>deactivated_edges(t_1, t_2)</code>	{}	{}	{}
	<code>changed_edges(t_1, t_2)</code>	{(<i>a, d</i>)}	{}	{}

$t_1 \leq t_2 \leq t'_2$. Finally, `next_activation(u, v, t)` finds the next activation time of edge $(u, v) \in E$ from time t . In order to retrieve the next activation, the query must satisfy $t \leq t'_1 \leq t'_2$. Note that an answer for this query can be t , some $t' > t$ or empty if this edge has no further activation.

Vertex-related queries retrieve vertices adjacent to a vertex at a time t or in a time interval $[t_1, t_2]$. The query `neighbors(u, t)` retrieves the vertices in the outgoing adjacency list of u at time t by finding all pairs $(u, v) \in E$ such that there is a contact (u, v, t) . Similarly, `neighbors(u, t_1, t_2)` retrieves the vertices in the outgoing adjacency list of u satisfying the weak or strong semantics. If it has weak semantics, an algorithm should retrieve all edges active at some timestamp in interval $[t_1, t_2]$ by finding all pairs $(u, v) \in E$ such that there is a contact (u, v, t') where $t_1 \leq t' \leq t_2$. Instead, if it has strong semantics, an algorithm retrieves only vertices adjacent to u active during all interval $[t_1, t_2]$ by finding all pairs $(u, v) \in E$ such that there is a persistent contact (u, v, t'_1, t'_2) and $t'_1 \leq t_1 \leq t_2 \leq t'_2$. Differently, `neighborsr(u, t)` and `neighborsr(u, t_1, t_2)` retrieve vertices in the incoming adjacency list of v at time t . In case of a directed graph, they retrieve vertices in the incoming adjacency list of v by finding all pairs $(u, v) \in E$, respectively, at time t or during an interval $[t_1, t_2]$, respecting weak or strong semantics.

Time-related queries retrieve edges, satisfying time constraints. For example, the query `aggregate(t)` retrieves the snapshot with time t by finding all edges $(u, v) \in E$ such that there is a contact (u, v, t) . The query `activated_edges(t)` retrieves edges $(u, v) \in E$ that became active at time t , *i.e.*, there is a contact (u, v, t) but not a contact $(u, v, t - 1)$. There is also the interval-based query `activated_edges(t_1, t_2)`. It retrieves all edges $(u, v) \in E$ that became active during $[t_1, t_2]$. Similarly, the queries `deactivated_edges(t)` and `deactivated_edges(t_1, t_2)` retrieve edges that became deactive at a time t or during an interval $[t_1, t_2]$. Finally, `changed_edges(t)` and `changed_edges(t_1, t_2)` retrieve edges that became active or deactive at time t or during an interval $[t_1, t_2]$.

Table 1 illustrates the low-level queries using the temporal graph presented in Figure 2. In the first column, we list the queries related to edge, vertex, and time; the second shows the results for point-time queries considering $t = 2$ (same as interval queries considering $t_1 = 3$ and $t_2 = 3$); and the third and fourth columns show the respective results for interval queries with weak and strong semantics considering the interval $[3, 5]$. For instance, $\text{neighbors}^r(d, 3, 3)$ (same as $\text{neighbors}^r(d, 3)$) retrieves the set $\{a\}$ with a single entry since (a, d) is the only direct edge that has some vertex incident to d and is activated at time $t = 3$. Differently, the query $\text{neighbors}^r(d, 3, 5)$ with weak semantic retrieves the set $\{a, e\}$ since edges (a, d) and (e, d) are active during any timestamp in interval $[3, 5]$. Finally, the query $\text{neighbors}^r(d, 3, 5)$ with strong semantic returns an empty set since there is no edge that has a vertex incident to d active during the whole interval $[3, 5]$.

2.2.2 High-level Queries for Temporal Graphs

High-level queries serve as the basis for graph analysis. Differently from low-level queries, these queries focus on extracting valuable knowledge in order to better understand the problem modeled by the graph. Some high-level queries solve the shortest path and connectivity problems in order to compare and find, for example, solutions for transportation problems (DING; GÜTING, 2004). Others compute rank measures such as degree centrality, closeness centrality, betweenness centrality, and PageRank, in order to discover important nodes in a social network (YANG et al., 2011). These tasks are computationally more demanding and need specialized strategies to reduce the time complexity. For example, one strategy is to distribute the computation in parallel to several machines using a vertex-centric approach (MALEWICZ et al., 2010). Another strategy is to use an edge-centric approach that exploits data locality to improve reading data from external memories (ROY; MIHAILOVIC; ZWAENPOEL, 2013). For detailed description on high-level queries and strategies to distribute the processing of these queries, we suggest the paper by Michail (2016).

Investigations on temporal reachability have been made for characterizing mobile and social networks (TANG et al., 2010; LINHARES et al., 2019); for validating protocols and better understanding communication networks (CACCIARI; RAFIQ, 1996; WHITBECK et al., 2012); for checking the existence of trajectories and improving flow in transportation networks (WU et al., 2017; WILLIAMS; MUSOLESI, 2016; BEDOGNI; FIORE; GLACET, 2018); for assessing future states of ecological networks (MARTENSEN; SAURA; FORTIN, 2017); and for making plans for agents using automation networks (BRYCE; KAMBHAMPATI, 2007). Beyond the sole reachability, some applications require the ability to reconstruct a concrete journey if one exists. For example, journey reconstruction has been used for finding and visualizing detailed

trajectories in transportation networks (WU et al., 2017; GEORGE; KIM; SHEKHAR, 2007; ZENG et al., 2014; HASAN et al., 2011); for visualizing system (HURTER et al., 2014) and infection spread dynamics (PONCIANO; VEZONO; LINHARES, 2021); and for matching temporal patterns in temporal graph databases (MOFFITT; STOYANOVICH, 2016; LATAPY; VIARD; MAGNIEN, 2018).

2.3 External Memory and Dynamic Data Structures

Recent technological advances such as the Internet of Things (IoT) and the integration of social media concepts in diverse applications have enabled new content to be created by anyone (ATZORI et al., 2012). Using high-capacity, long-term and cheaper storages is necessary in this context. We need algorithms and data structures well adapted to work with data on these storages. An effort towards recognizing the bottlenecks of current technologies can help us develop strategies that operate them optimally.

The Hard Disk Drive (HDD) technology is still the main long-term technology used to store a high amount of data with reasonably fast transfer rates. Storage technology manufactures have innovative plans for the upcoming years. For example, the Seagate company will soon produce devices with 120 TB capacity (SEAGATE, a; SEAGATE, b). Among the techniques to improve the cost benefit of HDDs are: the Shingled Magnetic Recording (SMR) (AMER et al., 2011), the Heat-Assisted Magnetic Recording (HAMR) (KRYDER et al., 2008), the Microwave-Assisted Magnetic Recording (MAMR) (ZHU; ZHU; TANG, 2008) and the Heated-Dot Magnetic Recording (HDMR) (HONO et al., 2018) in order to improve even more the cost benefit of HDDs. As long as HDDs maintain their price-capacity advantage relative to other memory technologies, they will continue to play a central role in computer applications.

HDDs are mechanical devices in which data can be read from or written to platters through magnetism (DATE, 2003). Several platters are stacked and each of their surfaces is used to store data. Each platter is divided into tracks, which are subdivided into sectors. A mechanical activator can move all the arms (one for each surface) holding read-write heads. To access a particular position, the activator must move the arm vertically, to get closer to the correct platter surface, then it moves the arm horizontally, to choose the correct track, finally the head accesses the correct sector when it is properly aligned during the next platter rotation. All this procedure for accessing the correct location on a platter is called *seek*. Data can also be transferred to a disk cache — a fast but small memory — in between read/write operations, commonly referred as I/O operations, to improve performance.

To read a file sequentially, a computer program must first open this file by making an `open` system call. This call seeks the beginning of the file on disk. Next, the program

must load the data from disk by making consecutive `read` system calls asking for the next chunk of data. During the execution of a `read` system call, the disk head transfers the corresponding data first to a local buffer and then to the space allocated in primary memory for the program. In fact, this operation transfers more data to the local buffer than it was asked for in order to take advantage of the current disk rotation, this is called *pre-fetch*, so during the next `read` system call the corresponding data will be already available in the local buffer. That is why it is so important to prioritize sequential I/O operations. As soon as the entire data of the current track is read, the activator moves the arm in direction of the next track. This process continues until the end of the file (`eof`) is reached.

Differently, to read from or write to a file at random locations (random I/Os), a computer program must first make explicit `seek` system calls and then make `read` or `write` system calls. If the seeked locations are very far from each other, the mechanical aspect of the HDD might severely degrade the program performance. The time to read the data is called *latency*, and it can be very high for random I/Os. That is another reason to prioritize sequential I/O operations. Nevertheless, the HDD tries to identify access patterns, using general policies, *e.g.*, the LRU policy (JOHNSON; SHASHA, 1994), and it writes data that is recurring asked for to a local cache. When the data corresponding to an I/O operation is already present in the cache, it is called a cache *hit*, otherwise it is called a cache *fault*.

Algorithms and data structures that work with large data on secondary memory must consider these aspects to operate slower storages optimally. As a general guide, first, they should try to reduce the number of random I/Os. For example, data can be stored in a compressed format to reduce the space it consumes on disk, and compact data structures can improve the space usage in primary memory. Second, they should take advantage of data locality by packing data that are commonly accessed together and then reading it sequentially. In Section 2.3.1, we present some dynamic data structures, such as the B-tree (BAYER; MCCREIGHT, 1970), that exploit data locality. Third, they should adapt the access pattern to the disk cache policies in order to maximize cache hits. We note it is not always possible since disk cache policies are developed for general purposes. Therefore, applications that have particular I/O patterns should implement a cache data structure in primary memory for their specific problem. Fourth, if more devices are available (because of their low cost), they should parallelize I/Os among all devices.

2.3.1 External Data Structures

External data structures support queries on large data sets. They can be static, when all data is processed once and queries are made after that, or dynamic, when queries

can be intermixed with update operations. In this document, we are more interested in dynamic data structures. For instance, dynamic data structures that perform *lookup*(x) queries while supporting the operations *insert*(x), *remove*(x) are very important, where x is some element. Next, we present two approaches commonly used to design such data structures: those based on hash tables and those based on trees.

2.3.1.1 Hash-based

Hash-based dynamic data structures usually have an amortized cost of $O(1)$ I/O operations for *insert*(x), *remove*(x) and *lookup*(x). In (FAGIN et al., 1979), the authors introduced a directory-based approach. Their data structure maintains a global array \mathcal{D} (the directory) containing 2^d pointers, for $d \geq 0$, that point to local buckets of size B that store the current elements. Multiple pointers may point to the same bucket. Then, they defined the function $\phi(x)$, which produces integers within the range $[0, \dots, K]$ for elements x and K sufficiently large, and $\phi_d(x)$, which returns the d least significant bits of $\phi(x)$. The function $\phi_d(x)$ maps elements x to locations in \mathcal{D} , and d is chosen such that it is the smallest number that guarantees at most B elements per bucket. Additionally, a value $k(\mathcal{B})$ is associated with every bucket \mathcal{B} , representing the number of the least significant bits shared by all elements in \mathcal{B} .

A simple algorithm answers *lookup*(x) using two I/O operations. First, it accesses $\mathcal{D}[\phi_d(x)]$, then it reads the bucket and checks whether x is present in it. Another algorithm performs *insert*(x) using an amortized cost of $O(1)$ I/O operations. First, it inserts x into the bucket \mathcal{B} pointed by $\mathcal{D}[\phi_d(x)]$. Then, in case \mathcal{B} becomes full, it splits \mathcal{B} while maintaining the data structure invariants. The split operation creates a new bucket \mathcal{B}' , then it shares the \mathcal{B} elements with \mathcal{B}' based on their $(k(\mathcal{B}) + 1)$ -th least significant bit, finally it assigns $k(\mathcal{B}) + 1$ to $k(\mathcal{B}')$ and $k(\mathcal{B})$. Finally, if $k(\mathcal{B}) > d$, the algorithm doubles the space of \mathcal{D} , then it initializes new pointers appropriately, and it increments d . Another algorithm performs *delete*(x) also having amortized cost of $O(1)$ I/Os. First, it removes x from the bucket \mathcal{B} pointed by $\mathcal{D}[\phi_d(x)]$. Then, in case \mathcal{B} becomes underflowed (less than 60% for example), it merges \mathcal{B} with the bucket \mathcal{B}' in which all elements share the same $k(\mathcal{B}) - 1$ least significant bits. Finally, it maintains the invariants accordingly.

The disadvantage of directory-based approaches is that the *lookup*(x) operation takes two I/O operations. In (LITWIN, 1980), the authors introduced a directory-less approach that answer *lookup*(x) in a single I/O. Their data structure maintains a global array \mathcal{L} containing the buckets $1, 2, \dots, 2^d + p - 1$, for $p < 2^d$, of size B . When the current limit of elements is reached, an algorithm creates a new bucket $2^p + p$, then it shares the elements of the bucket p with the bucket $2^p + p$ based on $\phi_{d+1}(x)$ values, and p is incremented by 1. If $p = 2^d$, then it resets p to 0 and d is incremented by 1. The downside of this strategy is that only the current bucket p can share elements, which may

not be the best option. The algorithm to answer $lookup(x)$ first computes $\phi_d(x)$, then, if $p \geq \phi_d(x)$, it simply reads the corresponding bucket and checks whether x is present in it, otherwise it reads the bucket corresponding to $\phi_{d+1}(x)$ instead (because the bucket $\phi_d(x)$ was split).

2.3.1.2 Tree-Based

Tree-based dynamic data structures usually have $O(\log_B N)$ I/O operations for $lookup(x)$, $insert(x)$ and $remove(x)$, where B is the size of a disk page. However, different from hash-based data structures, they maintain all elements sorted, so they additionally support range searches in an interval $[x, y]$ of ordered elements. Also, they support iterating orderly from a query result, *e.g.*, call $lookup(x)$ and iterate (orderly) until some element y .

The most used tree-based external data structure is the B-tree (BAYER; MCCREIGHT, 1970). A B-tree is a multi-way tree in which nodes can have up to B child pointers and $B - 1$ keys. Nodes must have at least $\frac{B}{2}$ children (except the root node) and data is ordered by a key. An algorithm that answers $lookup(x)$ descends the tree, similar to common Binary Search Trees (BSTs), by reading the next node and searching for x or the next child possibly containing x . It stops as soon as a node containing x is found, or it is no longer possible to find x . This time complexity of this algorithm is $O(\log_B N)$ because the B-tree height is $O(\log_B N)$, and it does one I/O operation per level visited.

An algorithm that performs $insert(x)$ similarly descends the tree and inserts x at the appropriate node \mathcal{N} . Then, in case \mathcal{N} becomes full, it splits \mathcal{N} and maintains the tree invariants. The split operation creates a new node \mathcal{N}' , then it shares half the elements of \mathcal{N} with \mathcal{N}' maintaining their order, and inserts a new entry into the parent of \mathcal{N} pointing to \mathcal{N}' . If the parent becomes full, it is split recursively upwards. Finally, if the root node is split, then it creates a new root node containing two children (the old root and its sibling) and the tree grows one level. The complexity is similar to the $lookup(x)$ query; however, its cost can be much higher because of the splits.

An algorithm that performs $remove(x)$ descends the tree and removes x from the corresponding node \mathcal{N} if x is found. Then, in case \mathcal{N} becomes underflowed, it tries to share the elements of the \mathcal{N} siblings with \mathcal{N} in order to restore the tree invariant. If no sibling can share, then it merges \mathcal{N} with one of its siblings, and removes from the parent the entry pointing to \mathcal{N} . If the parent becomes underflow, the same process is performed recursively upwards. Finally, if the root node becomes empty, then it removes the current root node, promotes the only child to be the new root, and the tree shrinks one level. Similarly, the complexity is similar to $insert(x)$.

Next, we list briefly some contributions that improved the original B-tree. In (ABEL, 1984), the authors introduced the B⁺-tree variant in which elements are

stored only in leaf nodes and each leaf node has a pointer to the next one. By doing so, elements can be easily iterated, as in a linked list. In (COMER, 1979), the authors proposed the B*-tree variant that instead of splitting node immediately, it first tries to share elements among both siblings. While B⁺-tree nodes have on average 69% of node utilization, the B*-tree nodes have on average 81% of utilization. In (AGARWAL; ERICKSON et al., 1999), the authors developed a variant that augments each node with a parent pointer. The naïve solution updates $\Theta(B)$ parent pointers during a split or a merge operation while their technique reduces it to $O(1)$. In (ARGE, 1995), the authors developed a variant that batches operations in node buffers in order to optimize I/Os. In (BECKMANN et al., 1990), the authors proposed the R-tree variant that generalizes the problem to multiple dimensions using a geometrical representation. Finally, in (CIACCIA; PATELLA; ZEZULA, 1997), the authors introduced the M-tree variant that generalizes the problem for objects in a metric space.

2.3.1.3 Dynamization of Static Data Structures

There are techniques that construct a collection of static data structures to simulate a dynamic data structure for specific problems. In (BENTLEY, 1979) the authors defined *decomposable search problems*. Formally, an arbitrary query $Q(x, S)$ with input x over a set of elements S is a decomposable search problem if we can answer it as $\bigcirc_{s \in S} Q(x, s)$ such that \bigcirc is a composition operator and s are partitions of S ; not necessarily unitary partitions. It means that we can decompose such problems in smaller ones, solve them individually, and combine their partial results to give a conclusive answer. For example, if $Q(x, S)$ is the query $Member(x, S)$, which answers whether $x \in S$, then we could partition S in smaller subsets, query each subset separately, and use the logical operator OR to compose those partial answers.

The authors also introduced a technique called the *logarithmic method* that solves dynamic decomposable search problems by maintaining static data structures and combining their answers. Their method maintains $O(\log N)$ static data structures of size 2^k , for $k \in [0, \log N - 1]$, such that there are no two data structures of the same size. Suppose we want to answer $lookup(N)$ queries and perform $insert(x)$ operations intermixed, however we only know how to construct static data structures for this problem using the operation $construct(x_1, x_2, \dots, x_N)$. By using the logarithmic method, an algorithm to perform $insert(x)$ would be as follows. Initially, given the operation $insert(x_1)$, it creates a data structure $s_1 = construct(\{x_1\})$ of size 1. Next, given $insert(x_2)$, it creates another data structure $s'_1 = construct(\{x_2\})$ of size 1, however, as s_1 and s'_1 have the same size, it creates a data structure $s_2 = construct(s_1 \cup s'_1)$ of size 2, and removes s_1 and s'_1 . By generalizing, when a new data structure of size 2^k is constructed and there is already another data structure of the same size, both are combined in order to create a new data structure of size 2^{k+1} . Then, at any moment, an algorithm that answers the $lookup(x)$ query simply

queries all current data structures and combines their partial results appropriately.

For the *lookup*(N) query, as there are up to $\log N$ static data structures of size at most $N/2$, its worst-case time complexity is $\log N$ times the cost of querying a single static data structure, which results in $O(\log^2 N)$ time for this case. For the *insert*(N) operation, as constructing data structures of size 2^k needs recombining $N/2^k$ smaller data structures, its worst-case time complexity is $O(2^k)$. Then, summing the construction of all static data structures, the total cost of maintaining N elements is $O(N \log N)$ time in the worst-case, and it is $O(\log N)$ amortized time for a single insertion. In (OVERMARS, 1987), the authors presented a technique in which single insertions are performed in $O(\log N)$ time in the worst-case.

Later, in (BENTLEY, 1979), the authors proposed other methods that give other trade-offs between query and update times by varying the number and sizes of the static data structures. Recently, in (COIMBRA et al., 2020), the authors used this technique to combine static compact representations for edge sets in order to propose a new dynamic data structure for graphs. There are also techniques that applied this approach to create new external dynamic data structures (AGARWAL; ERICKSON et al., 1999; ARGE; DANNER; TEH, 2004; ARGE; VITTER, 1996; AGARWAL et al., 2001). These techniques maintain only $\log_B N$ static data structures instead of $\log N$. Last, in (MATHIEU et al., 2021) the authors studied cases when queries and update operations have different distributions.

2.4 Concluding remarks

In this chapter we presented the fundamentals we used to study reachability queries on temporal graphs. First, we introduced concepts regarding temporal graphs, what is a temporal graph, how temporal reachability works in this type of graph, and some other queries a data structure for temporal graphs could support. Then, we presented general concepts about disk and described data structures to maintain data dynamically in secondary memory. For instance, B⁺-trees, one of the most used data structures to maintain sorted data in secondary memory, will be used later in this document to maintain collections of intervals in order to speed up reachability queries.

Related Work

We present in this chapter the related works we found in literature. In Section 3.1, we review studies on specialized data structures to answer reachability queries on temporal graphs. In 3.2, we review studies on disk-resident data structures to answer reachability queries on large temporal graphs. In Section 3.3, we review studies on compact data structures to store temporal graphs in primary memory. These data structures use minimal space while supporting common queries with (mostly) the same time complexity of non-compact data structures. Finally, in Section 3.4, we conclude this chapter with some considerations.

3.1 Reachability Queries

Deciding whether vertices can reach each other as the graph is continually modified is known as the dynamic reachability problem. The adjective *dynamic* does not refer to the temporal nature of networks, it refers to the fact that the computed information is to be updated after the input graph is changed. This maintenance is performed by a dynamic data structure, which stores intermediate information to speed up the query and update operations after a change (SCHAIK; MOOR, 2011; WANG et al., 2006; COHEN et al., 2003; ZHU et al., 2014; SEUFERT et al., 2013; WEI et al., 2018). Three types of dynamic data structures are classically considered, depending on the allowed changes, namely *incremental* (insertion only), *decremental* (deletion only), and *fully-dynamic* (both). Next, we summarize the techniques developed to solve these problems on (standard) undirected graphs, (standard) directed graphs, and temporal graphs, both in primary and secondary memories.

3.1.1 Reachability on Undirected Graphs

For undirected graphs (reachability example in Figure 4), it is enough to develop dynamic data structures that only consider reachability information in one direction, since

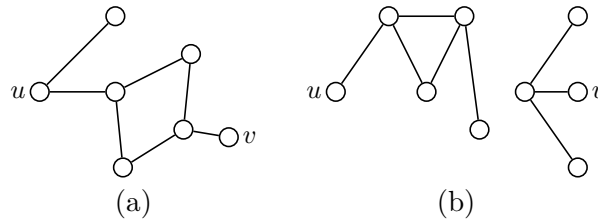


Figure 4 – Example of vertex reachability in undirected graphs. Figure (a) illustrates the case when there is a path from the vertex u to the vertex v . Figure (b) illustrates the case when there is no such path. We can easily check whether u can or cannot reach v in undirected graphs. We extracted both figures from (AJTAI; FAGIN, 1990).

the vertex relation is symmetric, *i.e.*, for $\forall u, v \in V$, if $u \rightarrow v$ (u reaches v through a path), then $v \rightarrow u$. Data structures can also compose information already computed as the vertex relation is also transitive, *i.e.*, for $\forall u, v, w \in V$, if $u \rightarrow w$ and $w \rightarrow v$, then $u \rightarrow v$. For example, in (TARJAN, 1979), the authors solved the incremental problem by maintaining connected components as sets of the union-find data structure (TARJAN, 1975). By doing so, update and query operations are computed in $\Theta(\alpha(n))$ time, where α is the inverse Ackermann function (ACKERMANN, 1928). Differently, (SHILOACH; EVEN, 1981) solved the decremental problem by maintaining Breath-First Searches (BFSs) as a layered tree structure with information about feasible paths in each of its layers. In their study, the authors proposed a parallel routine to update this layered structure along with the connected components associated with vertices in $O(n)$ amortized time, so their structure could answer reachability queries in constant time. Finally, (HOLM; LICHTENBERG; THORUP, 2001) solved the fully-dynamic problem by maintaining a spanning forest and a secondary structure holding the remaining edges that form cycles. The authors proposed algorithms to update both data structures in $O(\log^2 n)$ and achieved $O\left(\frac{\log n}{\log \log n}\right)$ time for reachability queries.

3.1.2 Reachability on Directed Graphs

For directed graphs (reachability example in Figure 5), data structures must consider reachability information in both directions, since the vertex relation is not symmetric, *i.e.*, there may exist a pair of vertices $(u, v) \in V$ such that $u \rightarrow v$ and $v \not\rightarrow u$. Nevertheless, they can still compose reachability information already computed, as this relation is transitive. Generally, solving many problems in directed graphs seems to be harder than in undirected graphs, as shown in (AJTAI; FAGIN, 1990). Specifically, the authors proved that the directed reachability problem is in a different complexity class from the same problem for the undirected case.

We found many studies on the static version of the directed reachability problem, in which input graphs are never modified. One common approach computes and com-

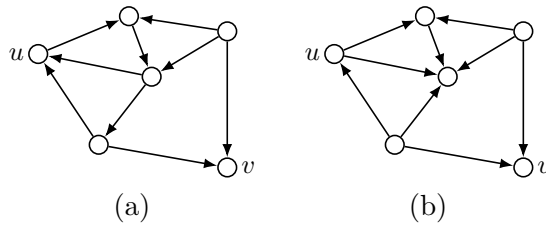


Figure 5 – Example of vertex reachability in directed graphs. Figure (a) illustrates the case when there is a path from u to v . Figure (b) illustrates the case when there is no such path. We cannot check as easily as in undirect graphs whether u can or cannot reach v . We extracted both figures from (AJTAI; FAGIN, 1990).

presses the Transitive Closure (TC) of the input graph in order to answer reachability queries as fast as possible (NUUTILA, 1995; SCHAIK; MOOR, 2011; CHEN; CHEN, 2008; CHEN; CHEN, 2011; JAGADISH, 1990; WANG et al., 2006). Techniques in this category usually store the TC information on disk because of its size (HIRVISALO; NUUTILA; SOISALON-SOININEN, 1996; AGRAWAL; BORGIDA; JAGADISH, 1989; YU; CHENG, 2010). For example, in (SCHAIK; MOOR, 2011), the authors introduced a space-efficient representation for TCs. First, their algorithm condenses the input graph by transforming its Strongly Connected Components (SCCs) into vertices and preserving edges connecting different SCCs. Then, it computes the TC of the condensed graph in $O(nm + n + m)$ time, where n is the number of vertices and m is the number of edges, by using a modified version of the Tarjan’s algorithm (TARJAN, 1972). Finally, it compresses each vertex’s reachability information into a list of intervals, in which each interval represents consecutive vertex ids. At query time, it is possible to answer reachability query in $O(\log l) \in O(\log n)$ time by searching the list of intervals of size l associated with the source vertex.

A different approach computes a vertex-labeling schema so that it can answer the queries using a fast-to-compute operator over the vertex labels (COHEN et al., 2003; WANG et al., 2015; CAI; POON, 2010; CHENG et al., 2008; JIN; WANG, 2013, 2013; CHENG et al., 2013; YANO et al., 2013). Techniques in this category use less space than the previous approach; however, the time to compute such schemas usually remains large. For example, in (COHEN et al., 2003) the authors introduced the 2-hop schema, in which each vertex u has associated labels $L_{in}(u)$ and $L_{out}(u)$ containing, respectively, some vertices that can reach u and some vertices reachable from u . At query time, to answer whether u reaches v , their algorithm simply checks whether $L_{out}(u) \cap L_{in}(v) \neq \emptyset$ in $O(|L_{out}(u)| + |L_{in}(v)|)$ time. The authors showed that their labeling schema uses $O(nm^{\frac{1}{2}})$ space, and that can be built in $O(n^3)$ time. Note that the selection of vertices in $L_{in}(u)$ and $L_{out}(v)$ must be careful, otherwise the query algorithm would produce false negatives (or false positives).

Another approach also computes a vertex-labeling schema to answer queries; how-

ever, the fast-to-compute operator can produce false negatives (or false positives) and, whenever this happens, it must fall back on search algorithms such as the BFS and the Depth-First Search (DFS) (SEUFERT et al., 2013; WEI et al., 2018; YILDIRIM; CHAOJI; ZAKI, 2012; CHEN; GUPTA; KURUL, 2005; VELOSO et al., 2014). Techniques in this category compute the additional information faster than other approaches, however queries can be much slower. For example, the algorithm introduced by Yildirim, Chaoji e Zaki (2012) performs d random walk traversals on the input graph to construct in $O(d(n+m))$ time a labeling schema of size $O(dn)$ based on lists of intervals. Their technique can produce false positives due to its randomized aspect, thus it can only compute queries quickly when the answer is negative. In the other case, it must perform a search algorithm on the graph. Their technique also uses the vertex labels as well to prune some unpromising paths. In contrast, the technique introduced in (SEUFERT et al., 2013) quickly computes queries whenever the answer is positive; the authors also showed that it outperforms the work by Yildirim, Chaoji e Zaki (2012) when queries are randomly produced.

We found some studies on the dynamic version of the directed reachability problem. However, most of them do not handle large graphs because of the larger cost of maintaining the additional data structures that are needed. For example, (ITALIANO, 1986) solved the incremental problem using the TC approach. The authors maintained a collection of spanning trees, each one having as root node one of the vertices, and a matrix, in which every cell (i, j) points to the node j of the spanning tree whose root node is i . Their algorithm for inserting an edge (u, v) merges, whenever necessary, the tree whose root node is v into the tree whose root node is u in $O(n)$ amortized time. Their query algorithm has constant time execution by simply checking whether the (u, v) cell is not *null*. Finally, their algorithm for reconstructing paths performs a bottom-up traversal from the node pointed by the (u, v) cell, where $k \leq n$ is the size of the resulting path.

In (ŁĄCKI, 2013), the authors tackled the decremental problem for directed graphs using the TC approach. In their paper, they presented first a solution that only works for DAGs. Instead of maintaining spanning trees, as in (ITALIANO, 1988), the authors maintained a DAG for each graph vertex. Note that a vertex v will become disconnected from a source vertex x of some DAG G only when the last edge (u, v) is deleted from G . By knowing that, the authors described the generic operation `find_unreachable_down`(G, S, w) (also used in the more general solution), where G is a DAG, S is a set containing source vertices, and w is a distinguished vertex. This operation returns a set U containing vertices that will be disconnected from w when deleting edges incident to S , and a set I containing all edges incident to U . The deletion algorithm runs as follows in $O(n)$ amortized time: given an edge (u, v) to be deleted, for each DAG G with root vertex x , call `find_unreachable_down`($G, \{v\}, x$) to retrieve (U, I) , then delete from G every vertex in U and every edge in I . The authors also introduced a new data structure

that maintains SCC information (with the same update cost) to solve the decremental reachability problem for directed graphs using $O(n + m)$ space while queries are done in constant time.

In (RODITTY, 2008), the author solved the fully-dynamic problem using also the TC approach. The technique presented in (RODITTY, 2008) improves the framework first proposed in (KING; SAGERT, 2002). In this framework, the insertion operation receives a set of edges incident to a vertex u as input, whereas the deletion operation receives an arbitrary set of edges. In (KING; SAGERT, 2002), the authors maintained a data structure that stores the SCC information of the current graph. Their update algorithm runs a linear time subroutine to detect the current SCCs and, by using this information, it can update by merging (during insertion) or by splitting (during deletion) the previous data structure information. However, in (RODITTY, 2008), the authors maintained a set of what they called *dynamic block*, which is a relaxation of the SCC definition. During an insertion operation, their algorithm constructs two trees centralized at u , a tree whose nodes can reach u and a tree whose nodes are reachable by u . During a deletion operation, their algorithm must maintain the graph edges and delete the appropriate edges from the previously built trees. These trees are important to merge and split dynamic blocks, whenever necessary, in order to provide reachability queries. By doing so, both their update algorithms have $O(n^2)$ amortized time complexity and reachability queries are done in constant time. Their data structure also supports reconstructing paths from source to target vertices in time proportional to the size of the resulting path.

Finally, other authors have studied other techniques based on the vertex-labeling schema approach and the vertex-labeling schema with fallback search approach (LYU et al., 2021; BRAMANDIA; CHOI; NG, 2008; RODITTY; ZWICK, 2016; ZHU et al., 2014). We note that usually these techniques prioritize the update time while providing a moderate query performance on average. Yet, other studies focus primarily on storing data structures for reachability queries on disk (HIRVISALO; NUUTILA; SOISALON-SOININEN, 1996; AGRAWAL; BORGIDA; JAGADISH, 1989; ZHANG et al., 2012b; ZHANG et al., 2018).

3.1.3 Reachability on Temporal Graphs

For temporal graphs (reachability example in Figure 6), data structures must always consider that vertices do not have a symmetric relation because the time dimension by itself imposes a direction. Unlike graphs, they cannot, so simply, compose the reachability information already computed as vertices do not have a transitive relation, *i.e.*, there may exist vertices $(u, v, w) \in V$ such that $u \rightsquigarrow v$ (u reaches v through a journey), $v \rightsquigarrow w$, but $u \not\rightsquigarrow w$; think when the first journey departs at time 10 and the second at time 5. Also, we cannot rely on strategies that try to maintain SCCs because, as proved

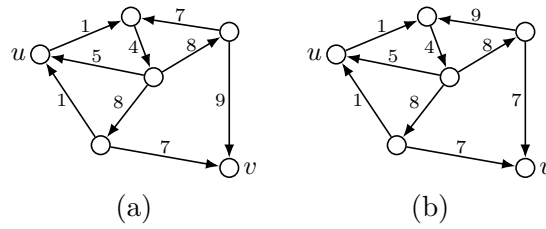


Figure 6 – Example of vertex reachability in temporal graphs. Figure (a) illustrates the case when there is a journey from u to v . Figure (b) illustrates the case when there is no such journey. We also cannot check so easily whether u can or cannot reach v ; we think it is more difficult to work with edges with timestamps because we also need to respect the journeys constraints.

by Casteigts (2018), the number of SCCs in a temporal graph can be exponential, when considering only non-strict journeys, or a super-polynomial, when considering only strict journeys.

The dynamic reachability problem can be further categorized based on whether modifying operations are chronologically ordered or not. This new categorization is being proposed in this thesis to better understand the scenarios. A solution for the chronological problem is useful in applications where contacts are continually inserted or deleted, but these updates do not follow a chronological order. For instance, during scenarios of epidemics, information containing the interaction details among infected and non-infected individuals is registered incrementally in a database. Then, this information is periodically queried in order to better understand the dissemination process and, thus, to support actions that slow it down or completely interrupt it (XIAO et al., 2018; XIAO; ASLAY; GIONIS, 2018). However, the collected data can arrive outdated, even though, sometimes, it is important to include this information for future analysis. Think when an infected patient goes to a hospital because he is feeling sick and the healthcare professional discovers that his infection occurred three days ago.

In the chronologically ordered case, only the latest snapshot of the temporal graph can be modified. For example, in (BARJON et al., 2014) the authors proposed algorithms for incrementally updating the reachability information in $O(\mu n \tau)$ time, where μ is the maximum number of edges present in a snapshot, in order to answer reachability queries in constant time. The central idea of their approach is to update the TC G_t^* at time t given the next snapshot G_{t+1} at time $t + 1$. In the non-chronologically ordered case, any snapshot of the temporal graph can be modified. Here, the solution proposed by Barjon et al. (2014) would not be suitable because G_t^* has not enough information to compute new journey possibilities (or impossibilities) after modifying a snapshot G_{t_2} such that $t_2 < t$.

To the best of our knowledge, (WU et al., 2016) is the only work that supports unsorted updates and exploits intermediate representations for speeding up reachability queries in temporal graphs. The authors introduced a solution to the fully-dynamic

problem using the vertex-labelling schema with search fallback approach. Their technique relies on maintaining a DAG in which every original vertex is possibly copied up to τ times (where τ is the number of timestamps) and a journey exists from u to v in the interval $[t, t']$ if and only if vertex u_t can reach vertex $v_{t'}$ in the DAG. However, the worst-case query time corresponds to a path search (*e.g.*, DFS) in the DAG, which takes $\Theta(n^2\tau)$ time for dense temporal graphs (whose number of contacts is of the same order). The space complexity (size of the DAG) also corresponds essentially to the number of contacts, thus $\Theta(n^2\tau)$ in the worst case. Finally, the update time upon insertion is quite efficient, because the DAG representation allows its effect to remain local. If one ignores the cost of paths preprocessing in (WU et al., 2016) (as we focus on worst-case analysis), it only takes $O(1)$ time to update the DAG if the corresponding vertices are already known, and up to $\Theta(\tau)$ otherwise, due to the creation of (up to) τ copies of the new vertices.

3.2 Reachability Queries on Disk

As discussed in Section 2.3.1, an external algorithm or data structure must take advantage of sequential I/Os while reducing the total number of I/Os. Next, we summarize some techniques developed to solve reachability queries for undirected and directed graphs and temporal graphs where data maintenance and processing are performed totally, or sometimes partially, on disk.

3.2.1 Reachability on Undirected Graphs Stored on Disk

First in (QIAO et al., 2012) and then in (QIAO, 2013), the authors proposed a static data structure to answer reachability queries on weighted undirected graphs. First, they proposed an in-memory solution to answer weight-constrained reachability queries in $O(1)$ time while consuming $O(|\Sigma||V|)$ space, where Σ is the weight set domain. Their technique computes the Minimum Spanning Tree (MST) and organizes its edges hierarchically so that the Least Common Ancestor (LCA) operation in this tree can be used to check whether there is a valid weight-constrained path in the input graph. Later, they introduced an external memory solution that answer reachability queries in $O(1)$ I/Os while consuming $O(|\Sigma||V|\log|V|)$ space. Their disk-resident technique rebalances the MST, to maintain a $\log|V|$ height, and encodes its nodes so that traversals are performed linearly on disk.

3.2.2 Reachability on Directed Graphs Stored on Disk

Zhang et al. (2012a) proposed a static data structure to answer reachability queries on directed graphs based on the vertex-labeling approach with BFS as search fallback algorithm. Different from (YILDIRIM; CHAOJI; ZAKI, 2012), which used an index that

can produce false negatives during the search stage to prune unpromising BFS branches, their technique computes an additional index that can produce false positives. These two indexes were combined to improve both the probability of answering queries quickly ($O(1)$ time) and the effectiveness of pruning unpromising BFS branches. Later, they proposed a solution to secondary memory using a partition-based heap in order to store and query both indexes on disk efficiently.

Differently, in (HIRVISALO; NUUTILA; SOISALON-SOININEN, 1996), the authors proposed a static data structure based on the TC approach. Their algorithm is a combination of three processing steps: the computation of the topological vertex order; the discovery of the SCCs by performing the Tarjan’s algorithm (TARJAN, 1972); and the construction of a list of successors for each vertex using the SCC information. In order to reduce space, they compressed the lists of successors by representing them as intervals, and, in order to reduce the number of I/Os, they developed a single-pass algorithm that combines all three processing steps while reading the input graph sequentially. They used the LRU policy (JOHNSON; SHASHA, 1994) to cache disk pages during the algorithm computation.

In (CHENG; YU; TANG, 2006), the authors proposed a dynamic data structure based on the vertex-labeling approach (with no false positives nor negatives). They based their technique on the 2-hop technique, which answers whether u reaches v by checking if $L_{out}(u) \cap L_{in}(v) \neq \emptyset$. This is the same as saying that an algorithm must find a vertex w contained by both $L_{out}(u)$ and $L_{in}(v)$. Their disk approach follows this later idea, and thus maintains a B⁺-Tree (ABEL, 1984) with keys being pairs (w, L) , where L are vertex labels used to filter down valid leaf nodes; and values being tables representing (u, v) pairs such that $L_{out}(u) \cap L_{in}(v) \neq \emptyset$. The authors suggest that their structure is easy to maintain, but they did not give any update algorithm due to space limit.

3.2.3 Reachability on Temporal Graphs Stored on Disk

In (SHIRANI-MEHR; KASHANI; SHAHABI, 2012), the authors proposed two static data structures to answer reachability queries on temporal graphs. The first one, named ReachGrid, groups contacts in a grid format in such a way that a search algorithm computes reachability queries by pruning unpromising paths related to impossible spatio and temporal expansions of events. On disk, grid blocks are pages so that related information is read sequentially. The second one, named ReachGraph, first transforms the input graph into a hyper-graph containing augmented nodes that contain vertices that can reach or are reachable by every other vertex inside other augmented nodes; then it pre-computes the reachability information between vertices in different hyper-nodes according to some pre-defined departures. During a query, the data structure is queried in order to answer reachability queries in $O(1)$ time; however, if the corresponding reachabil-

ity information is not available, a guided search algorithm must be performed. On disk, vertices that reach each other are placed in the same page. Furthermore, the hierarchy inside hyper-nodes along with the order of its timestamps is also considered to group information in disk pages.

In (STRZHELETSKA; TSOTRAS, 2017; STRZHELETSKA, 2018), the authors proposed static data structures to answer a slightly different reachability problem. Instead of considering that a contact occurs instantaneously at the specified time, they consider contacts are meetings and these meetings have durations. Therefore, these meeting durations must take into consideration when answering reachability queries.

3.3 Space-Efficient Data Structures for Querying Temporal Graphs in Primary Memory

These specialized data structures provide useful queries while spending little space as possible. Some of them store a compressed version of data. However, they compute queries without decompressing the whole data (BRISABOA et al., 2014; CARO; RODRÍGUEZ; BRISABOA, 2015; CARO et al., 2016; BRISABOA et al., 2018). The literature calls these approaches self-indexed space-efficient data structures.

For example, Grossi, Gupta e Vitter (2003) introduced the wavelet tree data structure that stores a list of contacts as a sequence of n symbols belonging to an alphabet of size $\sigma = |V| + |T|$ using only $n \lceil \log \sigma \rceil$ bits. The wavelet tree executes fundamental queries, such as determining the frequency of symbols in a sub-range of the sequence in $O(\log \sigma)$ time. By using the wavelet tree, some data structures can quickly answer low-level queries for temporal graphs. See Appendix A for an in-depth description of space-efficient data structures.

3.4 Concluding remarks

The literature about temporal graphs is growing rapidly, however little effort has been made to solve the dynamic reachability. If we consider the combination of all distinct problems in Section 3.1, *i.e.*, the combination of incremental, decremental and fully-dynamic data structures with chronologically and non-chronologically ordered operations, there are at least six different scenarios to be explored. There are also different approaches to solve the reachability problem that consider different trade-offs, *e.g.*, the TC approach, the vertex-labeling approach, etc. Therefore, we see a great horizon about reachability in temporal graphs to be yet discovered.

Also, little effort in literature was made to develop external algorithms and data

structures with performance comparable in complexity to the ones commonly designed for primary memory. For example, we did not find any study on dynamic data structures for answering reachability queries on temporal graphs. In fact, not even for the incremental or decremental case. Therefore, studies on this direction should be devised. The biggest challenge is to group data on disk so that algorithms can take advantage of sequential read.

A Dynamic Data Structure for Temporal Reachability with Unsorted Contact Insertions

In this chapter, we investigate the problem of maintaining an incremental data structure for temporal reachability, where the insertions of contacts are done in an arbitrary order. A naïve approach is to maintain the temporal graph as a set of contacts, then run standard journey computation algorithms like (XUAN; FERREIRA; JARRY, 2003a). However, the goal of data structures is to reduce the computational cost of the queries by pre-computing intermediate information. In fact, data structures typically offer a tradeoff between query time, update time, and space.

Our novel data structure¹ supports the four operations described in Chapter 1, that are, `add_contact(u, v, t)`, `can_reach(u, v, t1, t2)`, `is_connected(t1, t2)`, and `reconstruct_journey(u, v, t1, t2)`. The challenge in performing these operations is to answer queries as fast as possible, while keeping space consumption and update time at reasonable levels. The worst-case complexities of our algorithms are: the queries, `can_reach(u, v, t1, t2)` and `is_connected(t1, t2)`, run, respectively, in $O(\log \tau)$ and $O(n^2 \log \tau)$ time; the update operation, `add_contact(u, v, t)`, runs in $O(n^2 \log \tau)$ time; and the retrieval operation, `reconstruct_journey(u, v, t1, t2)`, runs in $O(k \log \tau)$ time, where n is the number of vertices, τ the number of timestamps, and $k < n$ is the length of the resulting journey. The worst-case space complexity remains within the size of the temporal graph itself, namely $O(n^2 \tau)$.

The core of our data structure is a component called the Timed Transitive Closure (TTC), which generalizes the classical notion of a Transitive Closure (TC). Classical TCs capture reachability information among vertices over the entire lifetime of the network.

¹ We have a simple implementation available at <https://github.com/albertiniufu/dynamictemporalgraph/>

They are classically encoded as a static directed graph where the existence of an edge from u to v implies that there is a journey u to v in the temporal graph. If one is not interested in querying reachability for specific subintervals, and if the contacts are inserted in chronological order, then TCs suffice for maintaining temporal reachability information (BARJON et al., 2014). A generalization of TC has also been considered in (WHITBECK et al., 2012), which allows queries to be parametrised by a maximum journey duration; however basic journey information, such as departures and arrivals, are not known, and the computation of the structure requires the information to be processed at once and chronologically (*i.e.*, subsequent updates are not supported).

In the unsorted (*i.e.*, non-chronological) case, TCs do not provide enough information to decide whether a new contact (possibly occurring at any point in history) can be composed with known journeys. To address this need, we introduce a generalization of TCs called TTCs, which store information regarding the availability of journeys for a well-chosen set of time intervals, without storing the journeys themselves. We study the general properties of TTCs, and we prove, in particular, that one can restrict the number of intervals considered to $O(\tau)$ for any pair of vertices (as opposed to $O(\tau^2)$), with immediate consequences on the space complexity of a data structure based on TTCs. This information is then exploited by our data structure algorithms.

The content present in this chapter was published on the Social Network Analysis and Mining journal (BRITO et al., 2022) available at <<https://link.springer.com/article/10.1007/s13278-021-00851-y>>.

We organize this chapter as follows. In Section 4.1, we introduce timed transitive closures, study their basic properties, and provide low-level primitives for manipulating them. In Section 4.2, we describe the algorithms that perform each operation of our data structure based on TTCs, together with their running time complexities. Finally, Section 4.3 concludes with some remarks and open questions.

4.1 Reachability Tuples and Timed Transitive Closure

In this section, we describe an extension of the concept of transitive closure called Timed Transitive Closure (TTC). The purpose of TTCs is to encode reachability information among the vertices, parametrised by time intervals, so that one can subsequently decide if a new contact occurring anywhere in history can be composed with existing journeys. The major components of TTCs are called *reachability tuples* (R-tuples). We introduce operators on R-tuples, such as inclusion and concatenation, and describe their role in the construction and maintenance of a TTC.

4.1.1 Reachability Tuples (R-tuples)

Just as the number of paths in a static graph, the number of journeys in a temporal graph could be too large to be stored explicitly (typically, factorial in n). To avoid this problem, R-tuples capture the fact that a vertex can reach another within a certain time interval without storing the corresponding journeys. Thus, a single R-tuple may capture the reachability information corresponding to many journeys. We distinguish between two versions of R-tuples, namely (*existential*) R-tuples and *constructive* R-tuples, the latter adding information for reconstructing a journey that witnesses reachability.

4.1.1.1 Existential R-tuples

The following definitions are given in the context of a temporal graph \mathcal{G} whose vertex set is V , lifetime is $\mathcal{T} = [1, \tau]$, and latency is δ .

Definition 4 (R-tuple). *An existential R-tuple is a tuple $r = (u, v, t^-, t^+)$, where u and v are vertices in \mathcal{G} , and t^- and t^+ are timestamps in \mathcal{T} . It encodes the fact that vertex u can reach vertex v through a journey \mathcal{J} such that $\text{departure}(\mathcal{J}) = t^-$ and $\text{arrival}(\mathcal{J}) = t^+$. If several such journeys exist, then they are all captured by the same R-tuple.*

The set of journeys captured by an R-tuple r is denoted by $\mathcal{J}(r)$, and we say that r *represents* these journeys. An R-tuple is *trivial* when it represents a trivial journey (*i.e.*, a single contact). Trivial R-tuples thus have the form $(u, v, t, t + \delta)$ for some t . The following relations and operations, compatible with any fixed value of parameter δ , are quite natural to define.

Definition 5 (Precedence \prec). *An interval $I_1 = [t_1^-, t_1^+]$ precedes an interval $I_2 = [t_2^-, t_2^+]$, denoted $I_1 \prec I_2$, if $t_1^+ \leq t_2^-$. Given two R-tuples $r_1 = (u_1, v_1, t_1^-, t_1^+)$ and $r_2 = (u_2, v_2, t_2^-, t_2^+)$, r_1 precedes r_2 , denoted $r_1 \prec r_2$ if $t_1^+ \leq t_2^-$ and $u_2 = v_1$.*

Intuitively, the precedence relation among R-tuples tells us that the journeys they represent can be composed, leading to another R-tuple.

Definition 6 (Concatenation \cdot). *Given two R-tuples $r_1 = (u_1, v_1, t_1^-, t_1^+)$ and $r_2 = (u_2, v_2, t_2^-, t_2^+)$ such that $r_1 \prec r_2$, the concatenation of r_1 with r_2 is the R-tuple $r_1 \cdot r_2 = (u_1, v_2, t_1^-, t_2^+)$.*

The natural inclusion among intervals extends to R-tuples as follows:

Definition 7 (Inclusion \subseteq). *Given two R-tuples $r_1 = (u_1, v_1, t_1^-, t_1^+)$ and $r_2 = (u_2, v_2, t_2^-, t_2^+)$, $r_1 \subseteq r_2$ if and only if $u_1 = u_2$, $v_1 = v_2$, and $[t_1^-, t_1^+] \subseteq [t_2^-, t_2^+]$ (that is, $t_2^- \leq t_1^- \leq t_1^+ \leq t_2^+$).*

If neither $r_1 \subseteq r_2$ nor $r_2 \subseteq r_1$ (or if the vertices are different), then r_1 and r_2 are called *incomparable*. Intuitively, if $r_1 \subseteq r_2$, then any of the journeys represented by r_2 could be replaced by a (possibly faster) journey represented by r_1 . More precisely:

Lemma 1. *Let u and v be two vertices in V . Let $\mathcal{I}_1 = [t_1^-, t_1^+]$ and $\mathcal{I}_2 = [t_2^-, t_2^+]$ be two subintervals of \mathcal{T} such that $\mathcal{I}_1 \subseteq \mathcal{I}_2$. If u can reach v within \mathcal{I}_1 , then u can reach v within \mathcal{I}_2 .*

Proof. The proof is straightforward, we give it for completeness. Let r be the R-tuple (u, v, t_1^-, t_1^+) and let \mathcal{J} be any of the journeys in $\mathcal{J}(r)$. One can reach v from u within \mathcal{I}_2 through the three following steps: (1) wait at u from t_2^- to t_1^- , (2) travel from u to v using \mathcal{J} , and finally (3) wait at v from t_1^+ to t_2^+ . \square

The consequence of Lemma 1 is that if $r_1 \subseteq r_2$, then r_2 is redundant for answering reachability queries from u to v .

Definition 8 (Redundancy). *Let S be a set of R-tuples and let $r \in S$, r is called redundant in S if there is $r' \in S$ such that $r' \subseteq r$. A set with no redundant R-tuple is called not-redundant.*

An R-tuple that is non-redundant in a set is also called *minimal* (in that set). It is natural to ask what the maximum size of an *not-redundant* set of R-tuples could be, with consequences for the space complexity of a reachability data structure based on R-tuples. It turns out that this number is always significantly smaller than the number of possible R-tuples.

Lemma 2. *The maximum size of a not-redundant set of R-tuples for \mathcal{G} is $\Theta(n^2\tau)$.*

Proof. First, we prove the maximum number of pair-wise incomparable R-tuples is $O(n^2\tau)$. Then, we show this bound is tight, as some graphs induce $\Theta(n^2\tau)$ incomparable R-tuples.

(1) *Upper bound:* There are $\Theta(n^2)$ ordered pairs of vertices. Thus, it suffices to show that for each pair (u, v) , the number of incomparable R-tuples whose starting vertex is u and whose ending vertex is v is $\Theta(\tau)$. Let S be a not-redundant set of such R-tuples, and let $r_1 = (u, v, t_1^-, t_1^+)$ and $r_2 = (u, v, t_2^-, t_2^+)$ be any two R-tuples in S . If $t_1^- = t_2^-$, then either $r_1 \subseteq r_2$ or $r_2 \subseteq r_1$, thus S is redundant (contradiction). As a result, all departures t_i^- belonging to the R-tuples in S are different, which implies that $|S| \leq \tau$.

(2) *Tightness:* Consider the complete temporal graph $\mathcal{K}_{n,\tau}$ on n vertices in which every edge is present in *all* timestamps in $[1, \tau]$. In such a graph, there are consequently $\Theta(n^2\tau)$ contacts, each of which is a trivial journey. Now, observe that either these journeys connect different vertices, or their intervals are incomparable (same duration with different starting times), thus none of them is redundant with the others. \square

Given a graph \mathcal{G} and a set S of R-tuples representing all the journeys of \mathcal{G} , the subset $S' \subseteq S$ of all minimal R-tuples is called the *representative* R-tuples of \mathcal{G} , denoted by $\mathcal{R}(\mathcal{G})$. We also write $\mathcal{R}(u, v)$ for those R-tuples in $\mathcal{R}(\mathcal{G})$ whose source is u and destination is v . From the proof of Lemma 2, we extract:

Observation 1. *Every contact of \mathcal{G} is present in $\mathcal{R}(\mathcal{G})$ as a trivial R-tuple.*

Observation 1 implies that $\mathcal{R}(\mathcal{G})$ is a *non-lossy* representation, as \mathcal{G} itself is contained in it. The downside is that its space complexity is at least as large as the number of contacts in \mathcal{G} . Observe that, up to a constant factor, it can however not be worse than the worst number of contacts, since there may exist up to $\Theta(n^2\tau)$ contacts, and not-redundant sets cannot exceed this size (Lemma 2). In other words, in dense temporal graphs, the reachability information offered by R-tuples is essentially free in space.

4.1.1.2 Constructive R-tuples

The data structure considered in this work has four operations, namely `add_contact`(u, v, t), `can_reach`(u, v, t_1, t_2), `is_connected`(t_1, t_2), and `reconstruct_journey`(u, v, t_1, t_2). The first three operations can be dealt with using only existential R-tuple. The fourth operation could benefit from storing a small amount of additional information into the R-tuple.

Definition 9 (Constructive R-tuple). *A constructive R-tuple is a tuple $r = (u, v, t^-, t^+, w)$ that contains the same information as an existential R-tuple, plus a vertex w such that at least one journey $\mathcal{J} \in \mathcal{J}(r)$ starts with the contact (u, w, t^-) . Vertex w is called the successor of u in r (resp., in \mathcal{J}).*

Most of the definitions and lemmas from Section 4.1.1 apply unchanged to constructive R-tuple. In particular, the definition of redundant R-tuples applies without considering the successor field. Indeed, if two constructive R-tuples differ only by the successor vertex, then they are equivalent and any of the two can be discarded. As for the concatenation of two constructive R-tuples $r_1 = (u_1, v_1, t_1^-, t_1^+, w_1)$ and $r_2 = (u_2, v_2, t_2^-, t_2^+, w_2)$, provided $r_1 \prec r_2$, we additionally require that the resulting R-tuple adopts the successor of r_1 as its own successor; that is, $r_1 \cdot r_2 = (u_1, v_2, t_1^-, t_2^+, w_1)$. For simplicity, whenever constructive R-tuples are not needed, we describe the algorithms using existential R-tuples.

4.1.2 Timed Transitive Closure

Informally, the timed transitive closure of a temporal graph \mathcal{G} is a multigraph that captures the existence of journeys within all time intervals, based on not-redundant R-tuples.

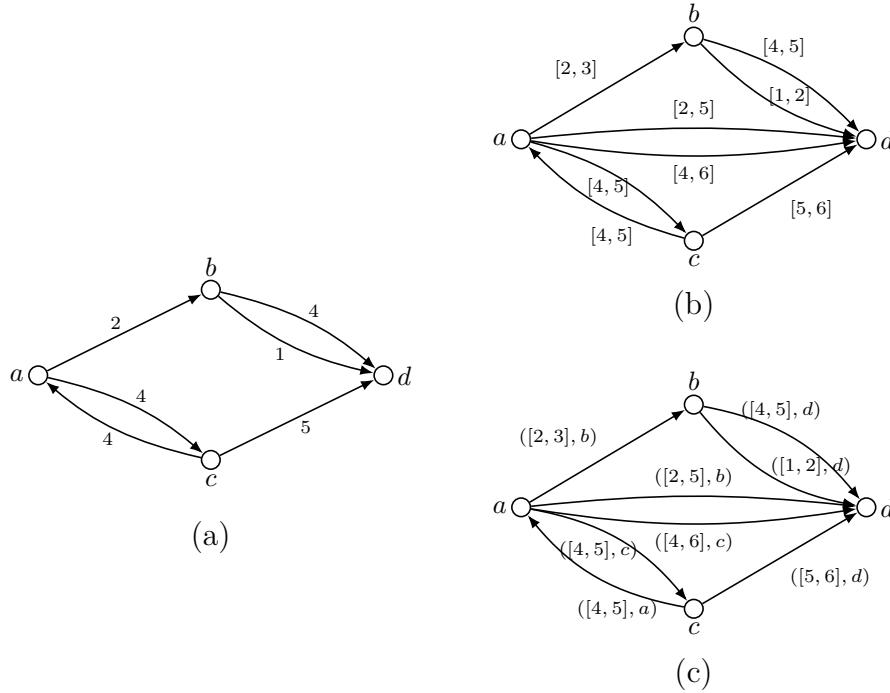


Figure 7 – $TTC(\mathcal{G})$ of the temporal graph \mathcal{G} on the left (a), considering $\delta = 1$. On the top right (b), the version with existential R-tuple, whose intervals are depicted by labels. On the bottom right (c), the version with constructive R-tuples, depicting also the successor.

Definition 10 (Timed transitive closure). *Given a graph \mathcal{G} , the timed transitive closure of \mathcal{G} , noted $TTC(\mathcal{G})$, is a (static) directed multigraph on the same set of vertices, whose edges correspond to the representative R-tuples of \mathcal{G} .*

Figure 7 shows two examples of TTCs (one for existential R-tuples, the other for constructive R-tuples). Algorithmically, a TTC provides most of the support needed to perform the high-level operations of our data structure. For example, the operation `can_reach`(u, v, t_1, t_2) can be answered by checking if there is an edge whose associated R-tuple is (u, v, t^-, t^+) with $[t^-, t^+] \subseteq [t_1, t_2]$. The operation `is_connected`(t_1, t_2) can be answered by performing such a test for every pair of vertices. The operation `add_contact`(u, v, t) reduces to adding a new edge to $TTC(\mathcal{G})$ if no smaller interval already captures this information. If the new edge is added, then some other edges may become redundant and should be removed, some others may also be created by composition. This operation is therefore the most critical. Finally, if constructive R-tuples are used, then an actual journey may be reconstructed efficiently from $TTC(\mathcal{G})$ when `reconstruct_journey`(u, v, t_1, t_2) is called, by retrieving a constructive R-tuple (u, v, t^-, t^+, w) such that $[t^-, t^+] \subseteq [t_1, t_2]$ and unfolding the corresponding journey inductively, by replacing u with the successor vertex w and t^- with $t^- + \delta$ in each step.

We describe all algorithms for these operations in Section 4.2. Before doing so, we present an explicit representation of TTCs based on adjacency matrices and Binary

Search Tree (BST). In order for the high-level algorithms to remain independent of this particular choice, we define a set of primitives for manipulating the TTC that are used by the high-level algorithms of Section 4.2.

4.1.2.1 Representing the TTC

We encode the TTC by an $n \times n$ matrix, in which every entry (i, j) points to a self-balanced BST denoted by $T_{(i,j)}$. The nodes in this tree contain all the time intervals corresponding to R-tuples in $\mathcal{R}(i, j)$. From Lemma 2, we know that a tree $T_{(u,v)}$ contains up to τ nodes. In addition, all these intervals are incomparable, thus one can use any of their boundaries (departure or arrival) as the sorting key of the BST. Note that retrieving $T_{(u,v)}$ within the matrix takes constant time, as the cells of a matrix are directly accessed. Also recall that finding the largest key below (resp. the smallest key above) a certain value takes $O(\log \tau)$ time. Similarly, inserting a new element (in our case, an interval) takes $O(\log \tau)$ time. Finally, observe that several types of BST (*e.g.*, red-black trees) can self-balance without affecting the asymptotic cost of insertions.

We provide the following low-level operations for manipulating TTCs: (1) $\text{FIND_NEXT}(T_{(u,v)}, t)$ returns the earliest interval $[t^-, t^+]$ in $T_{(u,v)}$ such that $t^- \geq t$, if any, and nil otherwise; symmetrically, (2) $\text{FIND_PREV}(T_{(u,v)}, t)$ returns the latest interval $[t^-, t^+]$ in $T_{(u,v)}$ such that $t^+ \leq t$, if any, and nil otherwise; finally, (3) $\text{INSERT}(T_{(u,v)}, t^-, t^+)$ inserts the interval $[t^-, t^+]$ in $T_{(u,v)}$ and performs some operations for maintaining the property that all intervals in $T_{(u,v)}$ are minimal.

Let us now describe the algorithms that perform these operations, along with their time complexities. The algorithm for $\text{FIND_NEXT}(T_{(u,v)}, t)$ searches $T_{(u,v)}$ recursively, by comparing t with the departure t^- of the current node interval $[t^-, t^+]$. If t^- is equal to or greater than t , then the current node is a candidate answer. The algorithm then compares the current node candidate and the previous one, and keeps the one containing the smallest (earliest) t^- , then it descends to the left child. Otherwise, if t^- is smaller than t , it simply descends to the right child. As soon as a leaf is reached (and visited), the algorithm returns the current candidate as the answer. The algorithm for $\text{FIND_PREV}(T_{(u,v)}, t)$ works symmetrically. The time complexities of both algorithms correspond to the depth of the tree, which is $O(\log \tau)$.

The algorithm for $\text{INSERT}(T_{(u,v)}, t^-, t^+)$ finds and removes any potential node with interval \mathcal{I}_i such that $[t^-, t^+] \subseteq \mathcal{I}_i$, then it inserts a new node containing $[t^-, t^+]$ using a standard BST insertion. Figure 8 gives a linear representation of the intervals in $T_{(u,v)}$ while performing this operation. A naïve implementation of this operation would consist of searching and removing each corresponding node independently. However, this would lead to a complexity of $O(d \log \tau)$ time, where d is the number of nodes removed, that is up to $O(\tau)$. We use a non-standard approach that makes it feasible in $O(\log \tau)$ time

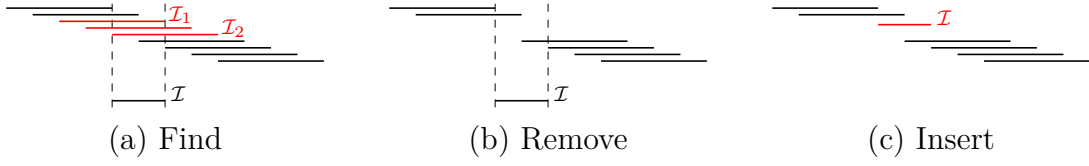


Figure 8 – Basic steps to perform $\text{INSERT}(T_{(u,v)}, t^-, t^+)$. First, in (a), an algorithm must find the candidate intervals that could become redundant after inserting $[t^-, t^+]$. These intervals are exactly the ones between $\mathcal{I}_1 = \text{FIND_NEXT}(T_{(u,v)}, t^+)$ and $\mathcal{I}_2 = \text{FIND_PREV}(T_{(u,v)}, t^-)$. Note that there are cases in which \mathcal{I}_1 or \mathcal{I}_2 do not exist. Next, in (b), all intervals \mathcal{I}' between (and including) \mathcal{I}_1 and \mathcal{I}_2 such that $[t^-, t^+] \subseteq \mathcal{I}'$ must be removed. Finally, in (c), the algorithm inserts $[t^-, t^+]$ in the correct place.

only. The strategy is to first identify in $T_{(u,v)}$ the nodes containing the boundary intervals \mathcal{I}_1 and \mathcal{I}_2 that correspond to the first and last nodes to be removed. The boundary interval \mathcal{I}_1 is found by calling $\mathcal{I}_1 = \text{FIND_PREV}(T_{(u,v)}, t^+)$, then checking whether \mathcal{I}_1 itself must be removed or not. If the arrival of \mathcal{I}_1 is smaller than t^+ , then either $[t^-, t^+]$ is redundant, and therefore the algorithm stops, or \mathcal{I}_1 should not be removed, and therefore \mathcal{I}_1 is replaced by the next node with greater key in $T_{(u,v)}$. Similarly, \mathcal{I}_2 is found by first calling $\mathcal{I}_2 = \text{FIND_NEXT}(T_{(u,v)}, t^-)$, then, if the departure of \mathcal{I}_2 is greater than t^- , \mathcal{I}_2 is replaced by the previous node with smaller keys in $T_{(u,v)}$. Note that the parameters passed to FIND_PREV and FIND_NEXT are indeed t^+ and t^- , not the reverse. Then, every node containing intervals in this range is removed using the technique outlined in the proof of Lemma 3.

Lemma 3. *In the worst case, the cost of the INSERT operation is $O(\log \tau)$.*

Proof. The range of intervals to be removed is characterized by two boundary intervals \mathcal{I}_1 and \mathcal{I}_2 , which can be found by calling both FIND_NEXT and FIND_PREV a single time, which takes $O(\log \tau)$ time. The final insertion of the input interval in the BST also takes $O(\log \tau)$. The difficult part is thus the removal of redundant intervals prior to this insertion (illustrated abstractly in Figure 8). Let $\text{SPLIT}(\mathcal{I})$ be the operation that splits a balanced BST into two balanced BSTs $T_{<\mathcal{I}}$ and $T_{\geq\mathcal{I}}$, where the first contains the intervals earlier than \mathcal{I} , and the second the intervals later or equal to \mathcal{I} . Let $\text{JOIN}(T_1, T_2)$ be the operation that receives as input two balanced BSTs and joins them into a single balanced BST. We proceed as follows. First, we split $T_{(u,v)}$ into two trees $T_{<\mathcal{I}_1}$ and $T_{\geq\mathcal{I}_1}$. Then, we split $T_{\geq\mathcal{I}_1}$ into two trees $T_{\geq\mathcal{I}_1 < \mathcal{I}_2}$ and $T_{\geq\mathcal{I}_2}$, and remove the smallest node of $T_{\geq\mathcal{I}_2}$ (by standard operations) to obtain $T_{>\mathcal{I}_2}$. The join of $T_{<\mathcal{I}_1}$ and $T_{>\mathcal{I}_2}$ is considered to be the new $T_{(u,v)}$. Indeed, this tree comprises all the original intervals except the ones in the range to be removed. Both the SPLIT and JOIN operations are known to be feasible at cost $O(\log \tau)$ on typical self-balanced trees, such as red-black trees (see (BLELLOCH; FERIZOVIC; SUN, 2016) for details). \square

Additionally, we define the following basic operations:

- $\mathcal{N}_{out}^*(u)$: Returns the set of vertices $\{v_1, v_2, \dots, v_k\}$ such that there is at least one edge from u to v_i in the TTC
- $\mathcal{N}_{in}^*(u)$: Returns the set of vertices $\{v_1, v_2, \dots, v_l\}$ such that there is at least one edge from v_i to u in the TTC

Both operations can be performed in $O(n)$ time, through traversing the corresponding row (resp. column) of the matrix and testing if the corresponding tree is empty.

4.2 The Four Operations

In this section, we describe the algorithms that perform the four operations of our data structure, whose contract was discussed in Chapter 1. These operations are `can_reach`(u, v, t_1, t_2), `is_connected`(t_1, t_2), `add_contact`(u, v, t), and (optionally) `reconstruct_journey`(u, v, t_1, t_2). For simplicity, the first three algorithms are presented using existential R-tuples only (however, they are straightforwardly adaptable to constructive R-tuples). All the algorithms rely on the primitives defined in Section 4.1.2.1 for manipulating the TTC abstractly. Then, we provide some experimental results regarding the average-case behavior of our data structure over contact insertions, and we formulate some open questions related to it.

4.2.1 Reachability and Connectivity Queries

The algorithm for `can_reach`(u, v, t_1, t_2) is straightforward. It consists of testing whether $T_{(u,v)}$ contains at least one interval that is included in $[t_1, t_2]$. This can be done by retrieving $[t^-, t^+] = \text{FIND_NEXT}(T_{(u,v)}, t_1)$ and checking that $t^+ \leq t_2$. Therefore, the cost of this algorithm reduces essentially to that of the operation `FIND_NEXT`($T_{(u,v)}, t_1$), which takes $O(\log \tau)$ time. We note that if $[t_1, t_2] = \mathcal{T}$ then it suffices to verify (in constant time) that $T_{(u,v)}$ is not empty. Regarding the operation `is_connected`(t_1, t_2), a simple way of answering it is to call `can_reach`(u, v, t_1, t_2) for every pair of vertices, with a resulting time complexity of $O(n^2 \log \tau)$. It seems plausible that this strategy is not optimal and could be improved.

4.2.2 Update Operation

The algorithm for `add_contact`(u, v, t) manages the insertion of a new contact (u, v, t) in the data structure, where $(u, v) \in E$ and $t \in \mathcal{T}$. To start, the interval corresponding to the trivial journey from u to v over $[t, t + \delta]$ is inserted in $T_{(u,v)}$ using the

INSERT primitive. Recall that this primitive encapsulates the removal of redundant intervals in $T_{(u,v)}$, if any. Then, the core of the algorithm consists of computing the indirect consequences of this insertion for the other vertices. Namely, if a vertex w^- could reach u before time t with the latest departure t^- and v could reach another vertex w^+ after time $t + \delta$ with the earliest arrival t^+ , it follows that w^- can now reach w^+ over interval $[t^-, t^+]$. Our algorithm consists of enumerating these compositions and inserting them in the TTC. Interestingly, for each predecessor w^- of u , only the *latest* interval ending before t in $T(w^-, u)$ needs to be considered. The reason is that in order to compose an earlier journey \mathcal{J} with the new contact, we need to wait at u until time t . Thus, even if some other journey started earlier, it would have to wait at u , and it would thus eventually arrive at the same time (based on a non-minimal interval). Based on this property, our algorithm only searches for the latest interval preceding t for each predecessor of u and the earliest interval exceeding $t + \delta$ for each successor of v .

The details are given in Algorithm 1, whose behavior is as follows. At line 1, the algorithm inserts the interval $[t, t + \delta]$ into $T_{(u,v)}$, which corresponds to the trivial journey induced by the new contact. From lines 2 to 7, for every vertex $w^- \in \mathcal{N}_{in}^*(u)$, it finds the latest interval $[t^-, _]$ in $T(w^-, u)$ that arrives before time t (inclusive) and inserts the composition $[t^-, t + \delta]$ into $T(w^-, v)$. For the same reasons as above, the algorithm only needs to consider inserting $[t^-, t + \delta]$ because every other composition would contain it as a subinterval. From lines 8 to 11, for every vertex $w^+ \in \mathcal{N}_{out}^*(v)$, the algorithm finds the earliest interval $[_, t^+]$ in $T(v, w^+)$ that leaves v after time $t + \delta$ (inclusive), and inserts the composition $[t, t^+]$ into $T_{(u,w^+)}$. In the same way, every other composition would contain $[t, t^+]$ as a subinterval. Finally, from lines 12 to 14, for all $w^- \in \mathcal{N}_{in}^*(u)$ and $w^+ \in \mathcal{N}_{out}^*(v)$, it inserts the composition $[t^-, t^+]$ into $T(w^-, w^+)$. In order to optimize this last step, the algorithm only considers the subset of \mathcal{N}_{in}^* whose reachability to v has been affected by the new contact, thanks to a dedicated storage D computed at line 7.

Theorem 4. *The update operation has worst-case time complexity $O(n^2 \log \tau)$.*

Proof. An INSERT operation is performed at line 1. The loop from line 3 to 7 iterates over $O(n)$ vertices and makes one insertion for each. The loop from line 8 to 14 iterates over $O(n)$ vertices, and for each one, iterates in a nested way over $O(n)$ vertices. For each resulting pair, it performs one INSERT operation. The latter clearly dominates the overall cost of the algorithm, with a cost of $O(n^2)$ times the cost of the INSERT operation, the latter being of time $O(\log \tau)$ (Lemma 3). \square

4.2.3 Journey Reconstruction

The algorithm for the operation `reconstruct_journey`(u, v, t_1, t_2) reconstructs a journey from vertex u to vertex v whose contact timestamps must be contained in $[t_1, t_2]$.

Algorithm 1 `add_contact`(u, v, t)

Require: $t \in \mathcal{T}, u, v \in V$ with $u \neq v$

```

1: INSERT( $T_{(u,v)}, t, t + \delta$ )
2:  $D \leftarrow \{\}$ 
3: for all  $w^- \in \mathcal{N}_{in}^*(u)$  do
4:    $[t^-, \_ ] \leftarrow \text{FIND\_PREV}(T_{(w^-,u)}, t)$ 
5:   if  $t^- \neq \text{nil}$  then
6:     INSERT( $T_{(w^-,v)}, t^-, t + \delta$ )
7:      $D \leftarrow D \cup (w^-, t^-)$ 
8: for all  $w^+ \in \mathcal{N}_{out}^*(v)$  do
9:    $[\_, t^+] \leftarrow \text{FIND\_NEXT}(T_{(v,w^+)}, t + \delta)$ 
10:  if  $t^+ \neq \text{nil}$  then
11:    INSERT( $T_{(u,w^+)}, t, t^+$ )
12:    for all  $(w^-, t^-) \in D$  do
13:      if  $w^- \neq w^+$  then
14:        INSERT( $T_{(w^-,w^+)}, t^-, t^+$ )

```

As explained in Section 4.1.1.2, existential R-tuples can be augmented by a *successor* field that indicates which vertex comes next in (at least one of) the journeys represented by the R-tuple. This information is very useful for reconstruction and has a negligible cost (asymptotically speaking). Concretely, one can make the nodes of the BST store the successor field besides the interval. The low-level operations for manipulating the TTC (see Section 4.1.2.1) are unaffected, nor are the query and update algorithms significantly. The only subtlety is that when two intervals (nodes) are composed, the successor field of the resulting node corresponds to the successor field of the first node (this was already discussed in terms of R-tuples in Section 4.1).

The goal of the algorithm is thus to reconstruct a journey by unfolding the intervals and successor fields. Details are given in Algorithm 2. The first step (from lines 1 to 3) is to

Algorithm 2 `reconstruct_journey`(u, v, t_1, t_2)

Require: $[t_1, t_2] \subseteq \mathcal{T}, u, v \in V, u \neq v$

```

1:  $([t^-, t^+], w) \leftarrow \text{FIND\_NEXT}(T_{(u,v)}, t_1)$  ▷ node augmented with successor
2: if the returned value is nil or  $t^+ > t_2$  then
3:   return nil ▷ no interval contained in  $[t_1, t_2]$  in  $T(u, v)$ 
4:  $\mathcal{J} \leftarrow \{(u, w, t^-\}$ 
5: while  $w \neq v$  do
6:    $([t, \_ ], w') \leftarrow \text{FIND\_NEXT}(T_{(w,v)}, t^- + \delta)$ 
7:    $\mathcal{J} \leftarrow \mathcal{J} \cdot \{(w, w', t)\}$ 
8:    $w \leftarrow w'$ 
9:    $t^- \leftarrow t$ 
10: return  $\mathcal{J}$ 

```

retrieve a node in $T(u, v)$ whose interval is contained within $[t_1, t_2]$ if one exists. If several choices exist, the earliest is selected (through calling the `FIND_NEXT` primitive). Then, the

algorithm iteratively replaces u with the successor and searches for the next interval until the successor is v itself (from lines 5 to 9), adding gradually the corresponding contacts to a journey \mathcal{J} (line 4 and line 7), which is ultimately returned at line 10.

Theorem 5. *Algorithm 2 has time complexity $O(k \log \tau)$, where k is the length of the resulting journey.*

Proof. The algorithm calls FIND_NEXT at line 1. After that, it is known whether a journey can be reconstructed. If so, a journey prefix \mathcal{J} is initialized with the first contact of the reconstructed journey (indeed, such a contact must exist because of the minimality of the interval). Then, in the loop from line 5 to line 9, the algorithm extends \mathcal{J} by one contact for each call to FIND_NEXT until \mathcal{J} contains the entire journey. Overall, FIND_NEXT is thus called as many times as the length of the reconstructed journey, which corresponds to $O(|\mathcal{J}| \log \tau)$ time. The costs of the other operations are clearly dominated by this cost. \square

4.2.3.1 Properties of the Reconstructed Journeys

Several journeys that satisfy the query parameters may exist. We observe that the specific choices made in Algorithm 2 imply additional properties.

Lemma 6. *The journey \mathcal{J} which is returned by Algorithm 2 is a foremost journey in the requested interval (i.e., it arrives at the earliest time at v). Among all the possible foremost journeys, it is also a fastest journey (i.e., the difference between departure and arrival is minimized).*

Proof. The fact that \mathcal{J} is a foremost journey follows from the call to FIND_NEXT at line 1. Indeed, the interval returned by this call corresponds to the earliest departure from u , which happens to also correspond to the earliest arrival at v because the stored intervals are incomparable. \mathcal{J} thus achieves the earliest arrival at v in the given interval. And since all the stored intervals are *minimal* (i.e., they do not contain smaller reachability intervals), it also follows that $\text{departure}(\mathcal{J})$ is as late as possible among all the journeys arriving in v at time $\text{arrival}(\mathcal{J})$, which means \mathcal{J} is as fast as possible among all foremost journeys. \square

Let us insist that Lemma 6 does not imply that \mathcal{J} is both foremost and fastest in the requested interval. It only states that \mathcal{J} is a foremost journey, and a fastest one *among* the possible foremost journeys. Even faster journeys might exist in the requested interval, arriving later at v . The above property is however already convenient, *e.g.*, in communication networks, where a message would arrive at the destination as early as possible, while (secondarily) traveling for as little time as possible.

4.2.4 Evolution of the Number of R-tuples over the Insertions

As explained in Lemma 2, the worst-case asymptotic number of R-tuples in the data structure cannot exceed the maximum number of contacts. The data structure is worst-case optimal in this regard. However, many typical scenarios involve contacts between only a small fraction of the possible pairs of vertices, and only at some specific times. In order to understand, more globally, how the data structure behaves at various densities of contacts, we investigated the evolution of the number of R-tuples, as new contacts are inserted, using a simple randomized model of contact insertions. Namely, we start with an empty data structure, then the contacts are drawn uniformly at random, without replacement, among a large set of pairs of vertices ($n = 100$) and of time steps ($\tau = 100$). The choice for these particular values is somewhat arbitrary, but we chose sufficiently large values to exhibit general phenomena that are discussed next. The experiments were performed for two values of latency δ , namely 0 and 1, which account for the distinction between strict and non-strict journeys. The results, averaged over 100 runs (in both cases), are shown in Figure 9 at different scales: (a) early evolution, (b) intermediate evolution, and (c) entire evolution.

Looking at the early evolution (Figure 9(a)), one can see that, at least for the first few insertions, the number of R-tuples seems to grow linearly with the number of contacts. A simple argument can explain this phenomenon: the first few contacts are *independent* of each other, in the sense that they share no vertices. As a result, these contacts do not combine into non-trivial journeys. This regime will change when the next contacts interact with the previous ones. Since every contact involves two vertices and every vertex is picked at random from a set of n vertices, the first non-trivial journey will be formed as soon as a vertex is picked twice. By the birthday paradox (GRINSTEAD; SNELL, 1997), this is expected to happen when $f(n) = 1 + \sum_{k=1}^n \frac{n!}{(n-k)!n^k}$ vertices are drawn, which corresponds to $f(n)/2$ contacts. Then, as further contacts are being inserted, the contacts combine with each other to form many more non-trivial journeys, as illustrated in Figure 9(b). When the number of inserted contacts reaches the theoretical maximum (i.e., the temporal graph becomes saturated), one can see in Figure 9(c) that the number of R-tuples converges to the number of contacts itself, as predicted (by Lemma 2).

Between the last two regimes, the behavior is different depending on whether $\delta = 0$ or $\delta = 1$. Interestingly, if $\delta = 1$, the number of R-tuples does not increase monotonically (see again Figure 9(c)). A plausible explanation for this phenomenon is as follows. When the contacts combine into non-trivial journeys, at first, almost none of these journeys improve upon existing ones (i.e. the new R-tuples do not *replace* existing ones, they just keep adding to the data structure). Then, as the number of existing R-tuples becomes huge, the insertion of a single new contact may induce a trivial R-tuple that will replace many redundant R-tuples (which are removed consequently). In the case that $\delta = 0$, the

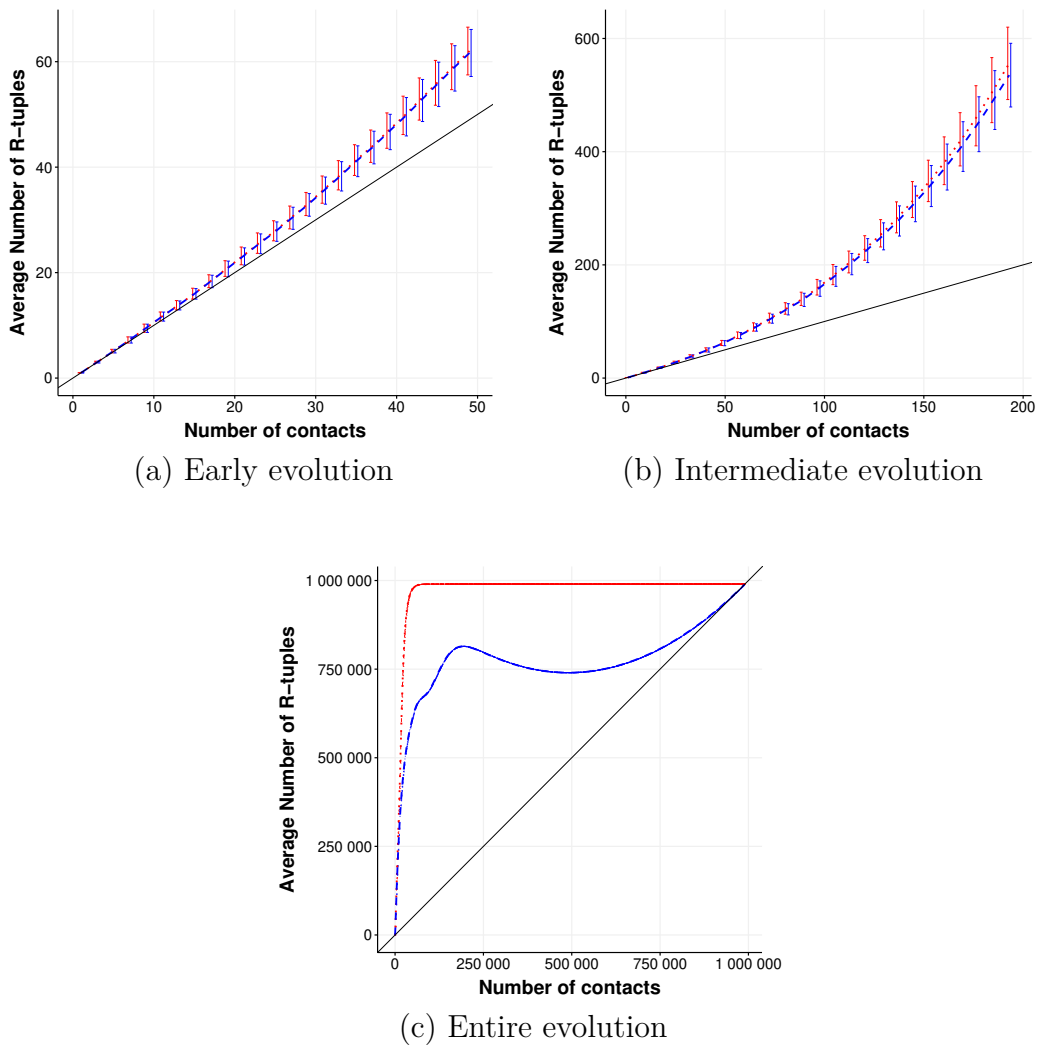


Figure 9 – Number of R-tuples stored in our data structure (in dotted red for $\delta = 0$ and dashed blue for $\delta = 1$), as a function of the number of inserted contacts (linear plot in plain black).

number of R-tuples reaches the theoretical maximum much faster, which can be explained by the fact that, as soon as a snapshot G_t becomes connected, then all the pairs of vertices in the graph now have a minimal R-tuple between them relative to time t . The minimality of these R-tuples also implies that these R-tuples will not be replaced when direct contacts appear between these pairs.

In summary, the evolution of the number of R-tuples seems to obey at least the following regimes, which are (1) linear; (2) superlinear; (3) decreasing (for $\delta = 1$); and (4) converging to saturation. There seems to be another phenomenon occurring between regimes (2) and (3) with $\delta = 1$, which would be interesting to investigate. In fact, we formulate three open questions to guide further investigations of these phenomena, which we think are of independent interest as they pertain to general aspects of temporal reachability.

Open question 1. *What is the worst-case theoretical growth of the second regime, depicted in Figure 9(b)? In particular, is it polynomial?*

Open question 2. *Is our explanation for the non-monotonicity of the number of R-tuples sufficient? Otherwise, what other mechanism explains it?*

Open question 3. *What phenomenon explains the mild inflexion that occurs with $\delta = 1$ (around 100000 insertions in Figure 9(c))?*

4.3 Concluding remarks

We presented in this chapter an incremental data structure to solve the dynamic connectivity problem in temporal graphs. Our data structure places a top priority on the query time, by answering reachability questions in time $O(\log \tau)$. Based on the ability to retrieve reachability information for particular time intervals, it supports the insertion of contacts in a non-chronological order in $O(n^2 \log \tau)$ worst-case time and makes it possible to reconstruct efficiently foremost journeys within a time interval, *i.e.*, in time $O(k \log \tau)$, where k is the size of the resulting journey. Our algorithms exploit the special features of non-redundant (minimal) reachability information, which we represent through the concept of R-tuples. The core of our data structure, namely the Timed Transitive Closure (TTC), is itself essentially a collection of not-redundant R-tuples, whose size (and that of the data structure itself) cannot exceed $O(n^2 \tau)$.

The theory of R-tuples poses several further questions, some of which are of independent interest, some leading to improvements in the presented algorithms. For example, do R-tuples involving different pairs of vertices possess further interdependence that may reduce the space needed to maintain TTCs? More generally, how restricted are TTCs intrinsically? On the practical side, can we improve the insertion time for new contacts by using another low-level structure than a balanced BST? Could the notion of contacts be generalized to contacts of arbitrary duration? Finally, designing efficient data structures for the decremental and the fully-dynamic versions of this problem, with *unsorted* contact insertion and deletion, seems to represent both a significant challenge and a natural extension of the present work, one that would certainly develop further our common understanding of temporal reachability.

A Dynamic Compact Data Structure for Temporal Reachability

In a computational environment, it is often useful to check whether entities can reach each other while using low space. As mentioned in Chapter 3, investigations on temporal reachability have been used, for instance, for characterizing mobile and social networks (TANG et al., 2010; LINHARES et al., 2019), and for validating protocols and better understanding communication networks (CACCIARI; RAFIQ, 1996; WHITBECK et al., 2012). Some other applications require the ability to reconstruct a concrete journey if one exists such as finding and visualizing detailed trajectories in transportation networks (WU et al., 2017; GEORGE; KIM; SHEKHAR, 2007; ZENG et al., 2014), and matching temporal patterns in temporal graph databases (MOFFITT; STOYANOVICH, 2016; LATAPY; VIARD; MAGNIEN, 2018). In all these applications, low space usage is important because it allows the maintenance of larger temporal graphs in primary memory.

In Chapter 4, we proposed a data structure that maintains a Timed Transitive Closure (TTC), a generalization of a TC that takes time into consideration. It maintains well-chosen sets of time intervals describing departure and arrival timestamps of journeys in order to provide time related queries and enable incremental updates on the data structure. The key idea is that, each set associated with a pair of vertices only contains non-nested time intervals and it is sufficient to implement all the TTC operations. Our previous data structure maintains only $O(n^2\tau)$ intervals (as opposed to $O(n^2\tau^2)$) using $O(n^2)$ dynamic Binary Search Trees (BSTs). Although the reduction of intervals is interesting, the space to maintain $O(n^2)$ BSTs containing $O(\tau)$ intervals each can still be prohibitive for large temporal graphs.

In this chapter, we propose a dynamic compact data structure to represent TTCs incrementally while answering reachability queries. Our new data structure maintains each set of non-nested time intervals as two dynamic bit-vectors, one for departure and

the other for arrival timestamps. Each dynamic bit-vector uses the same data layout introduced in (PREZZA, 2017), which resembles a B⁺-tree (ABEL, 1984) with static bit-vectors as leaf nodes. In this work, we used a raw bit-vector representation on leaves that stores bits as a sequence of integer words. In our experiments, we show that our new algorithms follows the same time complexities introduced in the previous section, however, the space to maintain our data structure is much smaller on temporally dense temporal graphs. Encoding (ELIAS, 1975) or packing (LEMIRE; BOYTSOV, 2015) the distance between 1's on leaves may improve the efficiency on temporally very sparse temporal graphs.

The content present in this chapter was published on the arXiv repository (BRITO et al., 2023) available at <https://arxiv.org/abs/2308.11734>.

We organize this chapter as follows. In Section 5.1, we briefly review the dynamic bit-vector proposed by Prezza (2017). In Section 5.2, we describe our new data structure along with the algorithms for each operation. In Section 5.3, we conduct some experiments comparing our new data structure with the data structure introduced in the previous chapter. Finally, Section 5.4 concludes with some remarks and open questions such as the usage of an encoding or packing techniques for temporally very sparse temporal graphs.

5.1 Dynamic bit-vectors

A bit-vector B is a data structure that holds a sequence of bits and provides the following operations: $\text{ACCESS}(B, i)$, which accesses the bit at position i ; $\text{RANK}_b(B, i)$, which counts the number of b 's until (and including) position i ; and $\text{SELECT}_b(B, j)$, which finds the position of the j -th bit with value b . It is a fundamental data structure to design more complex data structures such as compact sequence of integers, text, trees, and graphs (NAVARRO, 2016; CARO et al., 2016). Usually, bit-vectors are static, meaning that we first construct the data structure from an already known sequence of bits in order to take advantage of its query operations.

Additionally, a dynamic bit-vector allows changes on the underlying bits. Although many operations to update a dynamic bit-vector has been proposed, the following are the most commonly used: $\text{INSERT}_b(B, i)$, which inserts a bit b at position i ; $\text{UPDATE}_b(B, i)$, which writes the new bit b to position i ; and $\text{REMOVE}(B, i)$, which removes the bit at position i . Apart from these operations, there are others such as $\text{INSERT_WORD}_w(B, i)$, which inserts a word w at position i , and $\text{REMOVE_WORD}_n(B, i)$, which removes a word of n bits from position i .

In (PREZZA, 2017), the author proposed a dynamic data structure for bit-vectors with a layout similar to B⁺-trees (ABEL, 1984). Leaves wrap static bit-vectors of maximum length l and internal nodes contain at most m pointers to children along with the

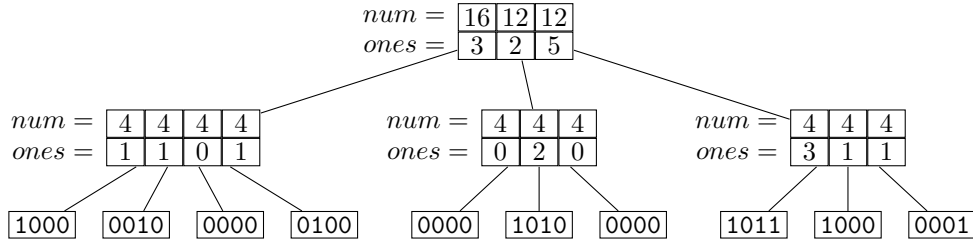


Figure 10 – A dynamic bit-vector using the data structure introduced in (PREZZA, 2017). Leaves wrap static bit-vectors and internal nodes contain pointers to children along with the number of 1’s and the total number of bits in each of them. The maximum number of pointers in each internal node m and the length of each static bit-vector n in this example is 4.

number of 1’s and the total number of bits in each subtree. With exception to the root node, static bit-vectors have a minimum length of $\lceil l/2 \rceil$ and internal nodes have at least $\lceil m/2 \rceil$ pointers to children. These parameters serve as rules to balance out tree nodes during insertion and removal of bits. Figure 10 illustrates the overall layout of this data structure.

Any static bit-vector representation can be used as leaves, the simplest one being arrays of words representing bits explicitly. In this case, the maximum length could be set to $l = \Theta(|w|^2)$, where $|w|$ is the integer word size. Other possibility is to represent bit-vectors sparsely by computing the distances between consecutive 1’s and then encoding them using an integer compressor such as Elias-Delta (ELIAS, 1975) or simply packing them using binary packing (LEMIRE; BOYTSOV, 2015). In this case, we can instead use as parameter the maximum number of 1’s encoded by static bit-vectors to balance out leaves.

Their data structure supports the main dynamic bit-vector operations as follows. An $\text{ACCESS}(B, i)$ operation is done by traversing the tree starting from the root node. In each visited node the algorithm searches from left to right for the branch that has the i -th bit and subtracts from i the number of bits in previous subtrees. After descending to the corresponding child node of this branch, the new i is local to that subtree and the search continues until reaching the leaf containing the i -th bit. At a leaf node, the algorithm simply accesses and returns the i -th local bit in the corresponding static bit-vector. If bits in static bit-vectors are encoded, an additional decoding step is necessary.

The $\text{RANK}_b(B, i)$ and $\text{SELECT}_b(B, j)$ operations are similar to $\text{ACCESS}(B, i)$. For $\text{RANK}_b(B, i)$, the algorithm also sums the number of 1’s in previous subtrees when descending to child nodes. At a leaf, it finally sums the number of 1’s in the corresponding static bit-vector up to the i -th local bit using `popcount` operations, which counts the number of 1’s in a word, and returns the resulting value. For $\text{SELECT}_b(B, j)$, the algorithm instead uses the number of 1’s in each subtree to guide the search. Thus, when traversing

down, it subtracts the number of 1's in previous subtrees from j , and sums the total number of bits. At a leaf, it searches for the position of the j -th local set bit using `clz` or `ctz` operations, which counts, respectively, the number of leading and trailing zeros in a word; sums it, and returns the resulting value.

The algorithm for $\text{INSERT}_b(B, i)$ first locates the leaf that contains the static bit-vector with the i -th bit. During this top-down traversal, it increments the total number of bits, and the number of 1's whether $b = 1$, in each internal node key associated with the child it descends. Then, it reconstructs the leaf while including the new bit b . If the leaf becomes full, the algorithm splits its content into two bit-vectors and updates its parent accordingly while adding a new key and a pointer to the new leaf. After this step, the parent node can also become full and, in this case, it must also be split into two nodes. Therefore, the algorithm must traverse back, up to the root node, balancing any node that becomes full. If the root node becomes full, then it creates a new root containing pointers to the split nodes along with the keys associated with both subtrees.

The algorithm for $\text{REMOVE}(B, i)$ also has a top-down traversal to locate and reconstruct the appropriate leaf, and a bottom-up phase to rebalance tree nodes. However, internal node keys associated with the child it descends must be updated during the bottom-up phase since the i -bit is only known after reaching the corresponding leaf. Moreover, a node can become empty when it has less than half the maximum number of entries. In this case, first, the algorithm tries to share the content of siblings with the current node while updating parent keys. If sharing is not possible, it merges a sibling into the current node and updates its parent while removing the key and pointer previously related to the merged node. If the root node becomes empty, the algorithm removes the old root and makes its single child the new root.

The $\text{UPDATE}_b(B, i)$ operation can be implemented by calling $\text{REMOVE}(B, i)$ then $\text{INSERT}_b(B, i)$, or by using a similar strategy with a single traversal.

5.2 Dynamic compact data structure for temporal reachability

Our new data structure uses roughly the same strategy as in the previous chapter. The main difference is the usage of a compact dynamic data structure to maintain sets of non-nested time intervals instead of Binary Search Trees (BSTs). This compact representation provides all BST primitives in order to incrementally maintain Timed Transitive Closures (TTCs) and answer reachability queries. In the previous chapter, we defined them as follows, where $T_{(u,v)}$ represents a BST holding a set of non-nested intervals associated with the pair of vertices (u, v) . (1) $\text{FIND_NEXT}(T_{(u,v)}, t)$ returns the earliest interval $[t^-, t^+]$ in $T_{(u,v)}$ such that $t^- \geq t$, if any, and nil otherwise; (2) $\text{FIND_PREV}(T_{(u,v)}, t)$ re-

turns the latest interval $[t^-, t^+]$ in $T_{(u,v)}$ such that $t^+ \leq t$, if any, and nil otherwise; and (3) $\text{INSERT}(T_{(u,v)}, t^-, t^+)$ inserts the interval $[t^-, t^+]$ in $T_{(u,v)}$ and performs some operations for maintaining the property that all intervals in $T_{(u,v)}$ are minimal.

For our new compact data structure, we take advantage that every set of intervals only contains non-nested intervals, thus we do not need to consider other possible intervals (Lemma 1). For instance, if there is an interval $\mathcal{I} = [4, 6]$ in a set, no other interval starting at timestamp 4 or ending at 6 is possible, otherwise, there would be some interval \mathcal{I}' such that $\mathcal{I}' \subseteq \mathcal{I}$ or $\mathcal{I} \subseteq \mathcal{I}'$. Therefore, we can represent each set of intervals as a pair of dynamic bit-vectors D and A , one for departure and the other for arrival timestamps. Both bit-vectors must provide the following low-level operations: $\text{ACCESS}(B, i)$, $\text{RANK}_b(B, i)$, $\text{SELECT}_b(B, j)$, $\text{INSERT}_b(B, i)$, and $\text{UPDATE}_b(B, i)$.

By using these simple bit-vectors operations, we first introduce algorithms for the primitives $\text{FIND_NEXT}((D, A)_{(u,v)}, t)$, $\text{FIND_PREV}((D, A)_{(u,v)}, t)$ and $\text{INSERT}((D, A)_{(u,v)}, t^-, t^+)$ that runs, respectively, in time $O(\log \tau)$, $O(\log \tau)$ and $O(d \log \tau)$, where d is the number of intervals removed during an interval insertion. Note that, now, these operations receive as first argument a pair containing two bit-vectors D and A associated with the pair of vertices (u, v) instead of a BST $T_{(u,v)}$. If the context is clear, we will simply use the notation (D, A) instead of $(D, A)_{(u,v)}$.

Then, in order to improve the time complexity of $\text{INSERT}((D, A)_{(u,v)}, t^-, t^+)$ to $O(\log \tau + d)$, we propose a new bit-vector operation: $\text{UNSET_ONE_RANGE}(B, j_1, j_2)$, which clears all bits in the range $[\text{SELECT}_1(B, j_1), \text{SELECT}_1(B, j_2)]$.

5.2.1 Compact representation of non-nested intervals

Each set of non-nested intervals is represented as a pair of dynamic bit-vectors D and A , one storing departure timestamps and the other arrival timestamps. Given a set of non-nested intervals $\mathcal{I}_1, \mathcal{I}_2, \dots, \mathcal{I}_k$, where $\mathcal{I}_i = [d_i, a_i]$, D contains 1's at every position d_i , and A contains 1's at every position a_i . Figure 11 depicts this representation.

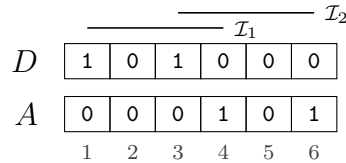


Figure 11 – Representation of a set of non-nested time interval using two bit-vectors, one for departures and the other for arrival timestamps. In this example, a set containing the intervals $[1, 4]$ and $[3, 6]$ is represented by the first bit-vector containing 1's at position 1 and 3, and the second bit-vector containing 1's at positions 4 and 6. Note that both bit-vectors must have the same number of 1's, otherwise, there would be an interval with missing values for departure or arrival.

5.2.2 Query algorithms

Algorithms 3 and 4 answers the primitives $\text{FIND_PREV}((D, A), t)$ and $\text{FIND_NEXT}((D, A), t)$, respectively. In order to find a previous interval, at line 1, Algorithm 3 first counts in j how many 1's exist up to position t in A . If $j = 0$, then there is no interval $I = [t^-, t^+]$ such that $t^+ \leq t$, therefore, it returns nil. Otherwise, at lines 4 and 5, the algorithm computes the positions of the j -th 1's in D and A to compose the resulting intervals. In order to find a next interval, at line 1, Algorithm 4 first counts in j' how many 1's exist up to time $t - 1$ in D . If $j' = \text{RANK}_1(D, \text{len}(D))$, then there is no interval $I' = [t'^-, t'^+]$ such that $t' \leq t^-$, therefore, it returns nil. Otherwise, at lines 4 and 5, the algorithm computes the positions of the $(j' + 1)$ -th 1's in D and A to compose the resulting interval.

Algorithm 3 $\text{FIND_PREV}((D, A), t)$

```

1:  $j \leftarrow \text{RANK}_1(A, t)$ 
2: if  $j = 0$  then
3:   return nil
4:  $t^- \leftarrow \text{SELECT}_1(D, j)$ 
5:  $t^+ \leftarrow \text{SELECT}_1(A, j)$ 
6: return  $[t^-, t^+]$ 

```

Algorithm 4 $\text{FIND_NEXT}((D, A), t)$

```

1:  $j \leftarrow \text{RANK}_1(D, t - 1)$ 
2: if  $j = \text{RANK}_1(D, \text{len}(D))$  then
3:   return nil
4:  $t^- \leftarrow \text{SELECT}_1(D, j + 1)$ 
5:  $t^+ \leftarrow \text{SELECT}_1(A, j + 1)$ 
6: return  $[t^-, t^+]$ 

```

As $\text{RANK}_1(B, i)$ and $\text{SELECT}_1(B, j)$ on dynamic bit-vectors have time complexity $O(\log \tau)$ using the data structure proposed by Prezza (2017), $\text{FIND_PREV}((D, A), t)$ and $\text{FIND_NEXT}((D, A), t)$ have both time complexity $O(\log \tau)$.

5.2.2.1 Interval insertion

Due to the property of non-containment of intervals, given a new interval $\mathcal{I} = [t_1, t_2]$, we must first assure that there is no other interval \mathcal{I}' in the data structure such that $\mathcal{I} \subseteq \mathcal{I}'$, otherwise, \mathcal{I} cannot be present in the set. Then, we must find and remove all intervals \mathcal{I}'' in the data structure such that $\mathcal{I}'' \subseteq \mathcal{I}$. Finally, we insert \mathcal{I} by setting the t_1 -th bit of bit-vector D and the t_2 -th bit of A . Figure 12 illustrates the process of inserting new intervals.

Algorithm 5 describes a simple process for the primitive $\text{INSERT}((A, D), t_1, t_2)$ in order to insert a new interval $\mathcal{I} = [t_1, t_2]$ into a set of non-nested intervals encoded

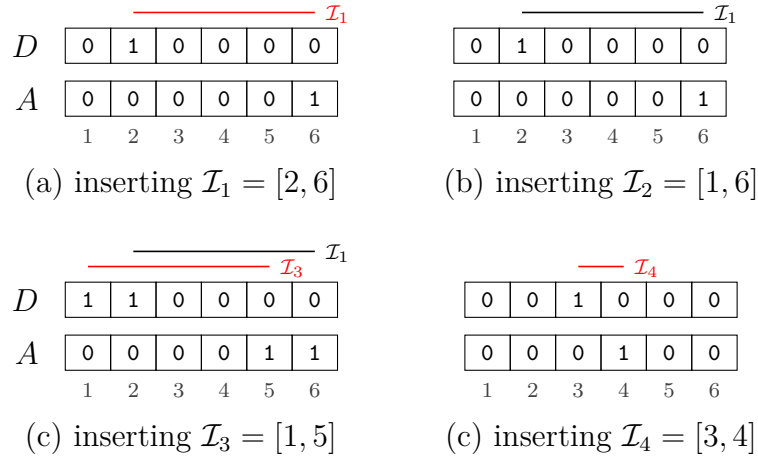


Figure 12 – Sequence of insertions using our data structure based on bit-vectors D and A . In (a), our data structure is empty, thus, the insertion of interval $\mathcal{I}_1 = [2, 6]$ results in setting the position 2 in D and 6 in A . Then, in (b), the new interval $\mathcal{I}_2 = [1, 6]$ encloses \mathcal{I}_1 , therefore, the insertion is skipped. Next, in (c), no interval encloses or is enclosed by the new interval $\mathcal{I}_3 = [1, 5]$, thus, it suffices to set the position 1 in D and 5 in A . Finally, in (d), the new interval $\mathcal{I}_4 = [3, 4]$ is enclosed by \mathcal{I}_1 and \mathcal{I}_3 , thus both of them is removed by clearing the corresponding bits and then \mathcal{I}_4 is inserted by setting the position 3 in D and 4 in A .

as two bit-vectors D and A . At line 1, it computes how many 1's exist in D prior to position t_1 by calling $r_d = \text{RANK}_1(D, t_1 - 1)$ and access the t_i -th bit in D by calling $bit_d = \text{ACCESS}(D, t_1)$. At line 2, it computes the same information with respect to the bit-vector A and timestamp t_2 by calling $r_a = \text{RANK}_1(A, t_2 - 1)$ and $bit_a = \text{ACCESS}(A, t_2)$. We note that the operations $\text{RANK}_1(B, i)$ and $\text{ACCESS}(B, i)$ can be processed in a single tree traversal using the dynamic bit-vector described in (PREZZA, 2017). If r_d is less than $r_a + bit_a$, then there are more intervals closing up to timestamp t_2 than intervals opening before t_1 , therefore, there is some interval $\mathcal{I}' = [d', a']$ such that $t_1 \leq d' \leq a' \leq t_2$, *i.e.*, $\mathcal{I} \subseteq \mathcal{I}'$. In this case, the algorithm stops, otherwise, it proceeds with the insertion. When proceeding, if $r_d + bit_d$ is greater than r_a , then there are more intervals opening up to t_1 than intervals closing before t_2 , therefore, there are $d = (r_d + bit_d) - r_a$ intervals $\mathcal{I}''_i = [d''_i, a''_i]$, such that $d''_i \leq t_1 \leq t_2 \leq a''_i$, *i.e.*, $\mathcal{I}''_i \subseteq \mathcal{I}$, that must be removed. From lines 5 to 9, the algorithm removes the d intervals that contain I by iteratively unsetting their corresponding bits in D and A . In order to unset the j -th 1 in a bit-vector B , we first search for its position by calling $i = \text{SELECT}_1(B, j)$, then update $B[i] = 0$ by calling $\text{UPDATE}_0(B, i)$. Thus, the algorithm calls $\text{UPDATE}_0(D, \text{SELECT}_1(D, r_a + 1))$ and $\text{UPDATE}_0(A, \text{SELECT}_1(A, r_a + 1))$ d times to remove the d intervals that closes after r_a . Finally, at lines 10 and 11, the algorithm inserts \mathcal{I} by calling $\text{UPDATE}_1(D, t_1)$ and $\text{UPDATE}_1(A, t_2)$. Note that both bit-vectors can grow with new insertions, thus we need to assure that both bit-vectors are large enough to accommodate the new 1's. That is why the algorithm calls *ensureCapacity*

before setting the corresponding bits. The *ensureCapacity* implementation may call $\text{INSERT}_0(B, \text{len}(B))$ or $\text{INSERT_WORD}_0(B, \text{len}(B))$ until B has enough space. Moreover, $\text{RANK}_1(B, i)$ and $\text{ACCESS}(B, i)$ operations can also receive positions that are larger than the actual length of B . In such cases, these operations must instead return $\text{RANK}_1(B, \text{len}(B))$ and 0, respectively.

Algorithm 5 $\text{INSERT}((D, A), t_1, t_2)$

```

1:  $r_d \leftarrow \text{RANK}_1(D, t_1 - 1)$ ;  $\text{bit}_d \leftarrow \text{ACCESS}(D, t_1)$ 
2:  $r_a \leftarrow \text{RANK}_1(A, t_2 - 1)$ ;  $\text{bit}_a \leftarrow \text{ACCESS}(A, t_2)$ 
3: if  $r_d \geq r_a + \text{bit}_a$  then
4:   if  $r_d + \text{bit}_d > r_a$  then
5:      $r_d^+ \leftarrow r_d + \text{bit}_d$ 
6:     while  $r_d^+ > r_a$  do
7:        $\text{UPDATE}_0(D, \text{SELECT}_1(D, r_a + 1))$ 
8:        $\text{UPDATE}_0(A, \text{SELECT}_1(A, r_a + 1))$ 
9:        $r_d^+ \leftarrow r_d^+ - 1$ 
10:   $\text{ensureCapacity}(D, t_1)$ ;  $\text{UPDATE}_1(D, t_1)$ 
11:   $\text{ensureCapacity}(A, t_2)$ ;  $\text{UPDATE}_1(A, t_2)$ 

```

Theorem 7. *The update operation has worst-case time complexity $O(d \log \tau)$, where d is the number of intervals removed.*

Proof. All operations on dynamic bit-vectors have time complexity $O(\log \tau)$ using the data structure proposed by Prezza (2017). As the maximum length of each bit-vector is τ , the cost of *ensureCapacity* is amortized to $O(1)$ during a sequence of insertions. Therefore, the time complexity of $\text{INSERT}((D, A), t_1, t_2)$ is $O(d \log \tau)$ since in the worst case Algorithm 5 removes d intervals from line 6 to 9 before inserting the new one at lines 10 and 11. \square

This simple strategy has a multiplicative factor on the number of removed intervals. In general, as more intervals in $[1, \tau]$ are inserted, the number of intervals d to be removed decreases, thus, in the long run, the runtime of this naïve solution is acceptable. However, when static bit-vectors are encoded sparsely as distances between consecutive 1's, it needs to decode/encode leaves d times and thus runtime degrades severely. In the next section, we propose a new operation for dynamic bit-vectors using sparse static bit-vectors as leaves, $\text{UNSET_ONE_RANGE}(B, j_1, j_2)$, to replace this iterative approach and improve the time complexity of $\text{INSERT}((D, A), t_1, t_2)$ to $O(\log \tau)$.

5.2.3 New dynamic bit-vector operation to improve interval insertion

In this section, we propose a new operation $\text{UNSET_ONE_RANGE}(B, j_1, j_2)$ for dynamic bit-vector using sparse static bit-vectors as leaves to improve the time com-

plexity of $\text{INSERT}((D, A), t_1, t_2)$. This new operation clears all bits starting from the j_1 -th 1 up to the j_2 -th 1 in time $O(\log \tau)$. Our algorithm for $\text{UNSET_ONE_RANGE}(B, j_1, j_2)$, based on the split/join strategy commonly used in parallel programs (BLELLOCH; FERIZOVIC; SUN, 2016), uses two internal functions $\text{SPLIT_AT_JTH_ONE}(N, j)$ and $\text{JOIN}(N_1, N_2)$. The $\text{SPLIT_AT_JTH_ONE}(N, j)$ function takes a root node N representing a dynamic bit-vector B and splits its bits into two nodes N_1 and N_2 representing bit-vectors B_1 and B_2 containing, respectively, the bits in range $[1, \text{SELECT}_1(B, j) - 1]$ and $[\text{SELECT}_1(B, j), \text{len}(B)]$. The $\text{JOIN}(N_1, N_2)$ function takes two root nodes N_1 and N_2 , representing two bit-vectors B_1 and B_2 and constructs a new tree with root node N representing a bit-vector B containing all bits from B_1 followed by all bits from B_2 . The resulting trees for both functions must preserve the balancing properties of dynamic bit-vectors (PREZZA, 2017).

Thus, given a dynamic bit-vector B represented as a tree with root N , our algorithm for $\text{UNSET_ONE_RANGE}(B, j_1, j_2)$ is described as follows. First, the algorithm calls $\text{SPLIT_AT_JTH_ONE}(N, j_1)$ in order to split the bits in B into two nodes N_{left} and N_{tmp} representing two bit-vectors containing, respectively, the bits in range $[1, \text{SELECT}_1(B, j_1) - 1]$ and in range $[\text{SELECT}_1(B, j_1), \text{len}(B)]$. Then, it calls $\text{SPLIT_AT_JTH_ONE}(N_{tmp}, j_2 - j_1)$ to split N_{tmp} further into two nodes N_{ones} and N_{right} containing, respectively the bits in range $[\text{SELECT}_1(B, j_1), \text{SELECT}_1(B, j_2) - 1]$, and $[\text{SELECT}_1(B, j_2), \text{len}(B)]$. The tree with root node N_{ones} contains all 1's previously in the original dynamic bit-vector B that should be cleared. In the next step, the algorithm creates a new tree with root node N_{zeros} containing $\text{len}(N_{ones})$ 0's to replace N_{ones} . Finally, it calls $\text{JOIN}(\text{JOIN}(N_{left}, N_{zeros}), N_{right})$ to join the trees with root nodes N_{left} , N_{zeros} , and N_{right} into a final tree representing the original bit-vector B with the corresponding 1's cleared.

Note that the tree with root N_{ones} is still in memory, thus it needs some sort of cleaning. The cost of immediately cleaning this tree would increase proportionally to the total number of nodes in N_{ones} tree. Instead, we keep N_{ones} in memory and reuse its children lazily in other operations that request node allocations so that the cost of cleaning is amortized. Moreover, even though we need to create a new bit-vector filled with zeros, this operation is performed in $O(1)$ time with a sparse implementation since only information about 1's is encoded. We do not recommend using this strategy for a dense implementation, i.e, leaves represented as raw sequences of bits, since this last operation would run in time $O(\tau)$.

Next we describe $\text{JOIN}(N_1, N_2)$ and $\text{SPLIT_AT_JTH_ONE}(N, j)$. The idea of $\text{JOIN}(N_1, N_2)$ is to merge the root of the smallest tree with the correct node of the highest tree and rebalance the resulting tree recursively.

Algorithm 6 details the $\text{JOIN}(N_1, N_2)$ recursive function. If $\text{height}(N_1) =$

Algorithm 6 JOIN(N_1, N_2)

```

1: if  $height(N_1) = height(N_2)$  then
2:   return  $mergeOrGrow(N_1, N_2)$ 
3: else if  $height(N_1) > height(N_2)$  then
4:    $R \leftarrow JOIN(extractRightmostChild(N_1), N_2)$ 
5:   if  $height(R) = height(N_1)$  then
6:     return  $mergeOrGrow(N_1, R)$ 
7:    $insertRightmostChild(N_1, R)$ 
8:   return  $N_1$ 
9: else
10:   $R' \leftarrow JOIN(N_1, extractLeftmostChild(N_2))$ 
11:  if  $height(R') = height(N_2)$  then
12:    return  $mergeOrGrow(R', N_2)$ 
13:   $insertLeftmostChild(N_2, R')$ 
14:  return  $N_2$ 

```

$height(N_2)$, at line 2, the algorithm tries to merge keys and pointers present in N_1 and N_2 if possible, or distributes their content evenly and grow the resulting tree by one level. This process is done by calling $mergeOrGrow(N_1, N_2)$, which returns the root node of the resulting tree. Instead, if $height(N_1) > height(N_2)$, at line 4, the algorithm first extracts the rightmost child from N_1 , by calling $extractRightmostChild(N_1)$, and then recurses further passing the rightmost child instead. The next recursive call might perform: a merge operation or grow the resulting subtree one level; therefore, the output node R may have, respectively, height equals to $height(N_1) - 1$ or $height(N_1)$. If the resulting tree grew, *i.e.*, $height(R) = height(N_1)$, then, at line 6, the algorithm returns the result of $mergeOrGrow(N_1, R)$. Otherwise, if a merge operation was performed, *i.e.*, $height(R) = height(N_1) - 1$, then, at line 7, it inserts R into N_1 as its new rightmost child, and returns N_1 . Finally, if $height(N_1) < height(N_2)$, at line 10, the algorithm extracts the leftmost child from N_2 by calling $extractLeftmostChild(N_2)$ and recurses further passing the leftmost child instead. Similarly, the root R' resulted from the next recursive call might have height equals to $height(N_2) - 1$ or $height(N_2)$. If $height(R') = height(N_2)$, then, at line 12, the algorithm returns the result of calling $mergeOrGrow(R, N_2)$, otherwise, if $height(R') = height(N_2) - 1$, then, at line 13, it inserts R' into N_2 as its new leftmost child, and returns N_2 . Note that all subroutines must properly update keys describing the length and number of 1's of the bit-vector represented by the corresponding child subtree. For instance, a call to $rightmost = extractRightmostChild(N)$ must decrement from the key associated with N the length and number of 1's in the bit-vector represented by $rightmost$.

Lemma 8. *The operation JOIN(N_1, N_2) has time complexity $O(|height(N_1) - height(N_2)|)$.*

Proof. Algorithm 6 descends at most $|height(N_1) - height(N_2)|$ levels starting from the root of the highest tree. At each level, in the worst case, it updates a node doing a constant amount of work equals to the branching factor of the tree. Therefore, the cost of $JOIN(N_1, N_2)$ is $O(|height(N_1) - height(N_2)|)$. \square

The idea of $SPLIT_AT_JTH_ONE(N, j)$ is to traverse N recursively while partitioning and joining its content properly until it reaches the node containing the j -th 1 at position $SELECT_1(B, j)$. During the forward traversal, it partitions the current subtree in two nodes N_1 and N_2 , excluding the entry associated with the child to descend. Then, during the backward traversal, it joins N_1 and N_2 , respectively, with the left and right nodes resulting from the recursive call.

Algorithm 7 $SPLIT_AT_JTH_ONE(N, j)$

```

1: if  $N$  is leaf then
2:    $(N_1, N_2) \leftarrow partitionLeaf(N, j)$ 
3:   return  $(N_1, N_2)$ 
4:  $(N_1, child, N_2) \leftarrow partitionNode(N, j)$ 
5:  $(N'_1, N'_2) \leftarrow SPLIT(child, j - ones(N_1))$ 
6: return  $(JOIN(N_1, N'_1), JOIN(N'_2, N_2))$ 

```

The details of this function is shown in Algorithm 7. From lines 1 to 3, the algorithm checks whether the root is a leaf. If it is the case, it partitions the current bit-vector $B_1 \cdot b \cdot B_2$, where b is the j -th 1, and returns two nodes containing, respectively, B_1 and $b \cdot B_2$. Otherwise, from lines 4 to 6, the algorithm first finds the i -th child that contains the j -th 1 using a linear search and partitions the current node into three other nodes: N_1 , containing the partition with all keys and children in range $[1, i - 1]$; $child$, which is the child node associated with position i ; and N_2 , containing the partition with all keys and children in range $[i + 1, \dots]$. Then, at line 5, it recursively calls $SPLIT_AT_JTH_ONE(child, j - ones(N_1))$ to retrieve the partial results N'_1 containing bits from $child$ up to the j -th 1; and N'_2 containing bits from $child$ starting at the j -th 1 and forward. Note that the next recursive call expects an input j that is local to the root node $child$. Finally, at line 6 it joins N_1 with N'_1 and N'_2 with N_2 , and returns the resulting trees.

Lemma 9. *The operation $SPLIT_AT_JTH_ONE(N, j)$ has time complexity $O(\log \tau)$.*

Proof. As $JOIN(N_1, N_2)$ has cost $O(|height(N_1) - height(N_2)|)$ and the sum of height differences for every level cannot be higher than the resulting tree height containing $n < \tau$ nodes, the time complexity of $SPLIT_AT_JTH_ONE(N, j)$ is $O(\log \tau)$. \square

Furthermore, since $JOIN(N_1, N_2)$ outputs a balanced tree when concatenating two already balanced trees, both trees resulting from the $SPLIT_AT_JTH_ONE(N, j)$ calls are also balanced.

Lemma 10. *The operation $\text{UNSET_ONE_RANGE}(B, j_1, j_2)$ has time complexity $O(\log \tau)$ when B encodes leaves sparsely.*

Proof. The $\text{UNSET_ONE_RANGE}(B, j_1, j_2)$ operation calls SPLIT_AT_JTH_ONE and JOIN twice. It must also create a new tree containing $\text{SELECT}_1(B, j_2 - 1) - \text{SELECT}_1(B, j_1)$ 0's to replace the subtree containing $j_2 - j_1$ 1's. If leaves of B are represented sparsely, then the creation of a new tree filled with 0's costs $O(1)$ since the resulting tree only has a root node, with its only key having the current length ($\text{SELECT}_1(B, j_2 - 1) - \text{SELECT}_1(B, j_1)$), and an empty leaf. Therefore, as the cost of $\text{SPLIT_AT_JTH_ONE}(N, j)$, $O(\log \tau)$, dominates the cost of $\text{JOIN}(N_1, N_2)$, the time complexity of $\text{UNSET_ONE_RANGE}(B, j_1, j_2)$ is $O(\log \tau)$. \square

Theorem 11. *The primitive $\text{INSERT}((D, A), t^-, t^+)$ has time complexity $O(\log \tau)$ when D and A encode leaves sparsely.*

Proof. Following from Theorem 7 and Lemma 10, the loop in Algorithm 5 that iteratively unset d bit-vector bits can be substituted by a call to $\text{UNSET_ONE_RANGE}(B, j_1, j_2)$. As the cost of Algorithm 5 is dominated by this loop, its time complexity reduces to $O(\log \tau)$. \square

5.3 Experiments

In this section, we conduct experiments to analyze the wall-clock time performance and the space efficiency of data structures when adding new information from synthetic datasets. In Section 5.3.1, we compare our compact data structure that maintain a set of non-nested intervals directly with an in-memory B^+ -tree implementation storing intervals as keys. For our compact data structure, we used dynamic bit-vectors (PREZZA, 2017) with leaves storing bits explicitly as arrays of integer words with words being 64 bits long. Internal nodes have a maximum number of pointers to children $m = 32$ and leaf nodes have static bit-vectors with maximum length $l = 4096$. For the B^+ -tree implementation we used $m = 32$ for all nodes. In Section 5.3.2, we compare the overall Timed Transitive Closures (TTCs) data structure using our new compact data structure with the TTC using the B^+ -tree implementation for each pair of vertices. All code is available at <https://bitbucket.org/luizufu/zig-ttc/src/master/>.

5.3.1 Comparison of data structures for sets of non-nested intervals

For this experiment, we created datasets containing all $O(\tau^2)$ possible intervals in $[1, \tau]$ for $\tau \in [2^3, 2^{14}]$. Then, for each dataset, we executed 10 times a program that

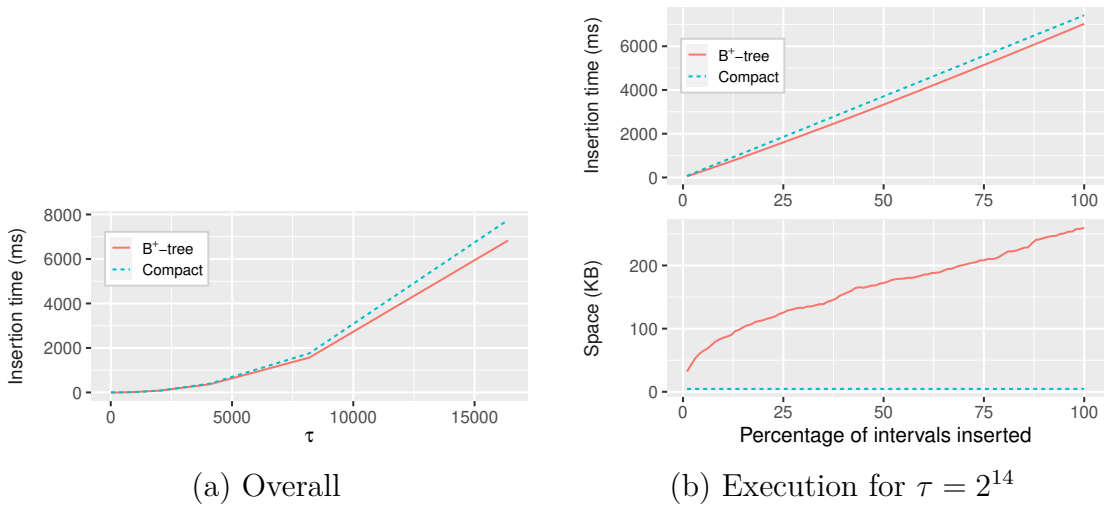


Figure 13 – Comparison of incremental data structures to represent a set of non-nested intervals. In (a), the overall average wall-clock time to insert all possible $O(\tau)$ intervals randomly shuffled into data structures. In (b), the cumulative wall-clock time and the memory space usage to insert all possible $O(\tau)$ intervals randomly shuffled throughout a single execution. Note that the final wall-clock time of the execution described in (b) was one of the 10 executions with $\tau = 2^{14}$ used to construct (a).

shuffles all intervals at random, and inserts them into the tested data structure while gathering the wall-clock time and memory space usage after every insertion.

Figure 13(a) shows the average wall-clock time to insert all intervals into the both data structures as τ increases. Figure 13(b) shows the cumulative wall-clock time to insert all intervals and the memory usage throughout the lifetime of a single program execution with $\tau = 2^{14}$. As shown in Figure 13(a), our new data structure slightly underperforms when compared with the B⁺-tree implementation. However, as shown in Figure 13(b), the wall-clock time have a higher overhead at the beginning of the execution (first quartile) and, after that, the difference between both data structures remains almost constant. This overhead might be due to insertions of 0's at the end of the bit-vectors in order to make enough space to accommodate the rightmost interval inserted so far. We can also see in Figure 13(b) that the space usage of our new data structure is much smaller than the B⁺-tree implementation. It is worth noting that, if the set of intervals is very sparse, maybe the use of sparse bit-vector as leaves could decrease the space since it does not need to preallocate most of the tree nodes, however, the wall-clock time could increase since at every operation leaves need to be decoded/unpacked and encoded/packed.

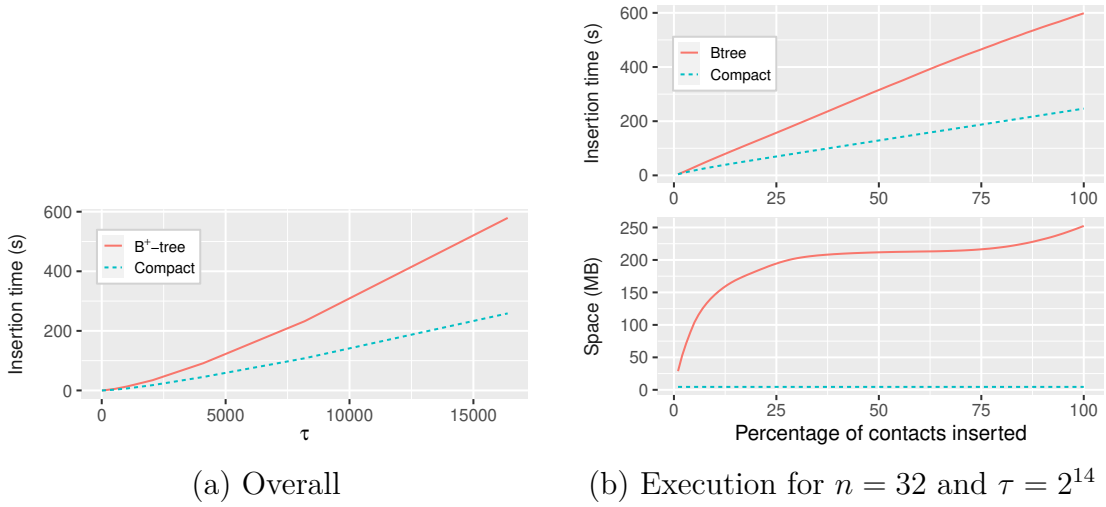


Figure 14 – Comparison of Timed Transitive Closures (TTCs) using incremental data structures to represent sets of non-nested intervals for each pair of vertices. In (a), the overall average wall-clock time to insert all possible $O(n^2\tau)$ contacts randomly shuffled into data structures. In (b), the cumulative wall-clock time and the memory space usage to insert all possible $O(n^2\tau)$ contacts randomly shuffled throughout a single execution. Note that the final wall-clock time of the execution described in (b) was one of the 10 executions with $\tau = 2^{14}$ used to construct (a).

5.3.2 Comparison of data structures for Time Transitive Closures

For this experiment, we created datasets containing all $O(n^2\tau)$ possible contacts fixing the number of vertices $n = 32$ and the latency to traverse an edge $\delta = 1$ while varying $\tau = [2^3, 2^{14}]$. Then, for each dataset, we executed 10 times a program that shuffles all contacts at random, and inserts them into the tested TTC data structure while gathering the wall-clock time and memory space usage after every insertion.

Figure 14(a) shows the average wall-clock time to insert all contacts into the TTCs using both data structures as τ increases. Figure 14(b) shows the cumulative wall-clock time to insert all contacts and the memory usage throughout the lifetime of a single program execution with $n = 32$ and $\tau = 2^{14}$. As shown in Figure 14(a), the TTC version that uses our compact data structure in fact outperforms when compared with TTC that uses the B⁺-tree implementation for large values of τ . In Figure 14(b), we can see that the time to insert a contact into the TTC using our new data structure is lower during almost all lifetime, and the space usage followed the previous experiment comparing data structures in isolation.

5.4 Concluding remarks

We presented in this chapter an incremental compact data structure to represent a set of non-nested time intervals. This new data structure is composed by two dynamic bit-vectors and works well using common operations on dynamic bit-vectors. Among the operations of our new data structures are: $\text{FIND_PREV}((A, D), t)$, which retrieves the previous interval $[t_1, t_2]$ such that $t_2 \leq t$ in time $O(\log \tau)$; $\text{FIND_NEXT}((A, D), t)$, which retrieves the next interval $[t_1, t_2]$ such that $t_1 \geq t$ also in time $O(\log \tau)$; and $\text{INSERT}((A, D), t_1, t_2)$, which inserts a new interval $\mathcal{I} = [t_1, t_2]$ whether no other interval \mathcal{I}' such that $\mathcal{I} \subseteq \mathcal{I}'$ exists while removing all intervals \mathcal{I}'' such that $\mathcal{I} \subseteq \mathcal{I}''$ in time $O(d \log \tau)$, where d is the number of intervals removed. Moreover, we introduced a new operation $\text{UNSET_ONE_RANGE}(B, j_1, j_2)$ for dynamic bit-vectors that encode leaves sparsely, which we used to improve the time complexity of our insert algorithm to $O(\log \tau)$.

Additionally, we used our new data structure to incrementally maintain Timed Transitive Closures (TTCs) using much less space. We used the same strategy as described in the previous chapter, however, instead of using BSTs, we used our new compact data structure. The time complexities of our algorithms for the new data structure are the same as those for BSTs. However, as we showed in our experiments, using our new data structure greatly reduced the space usage for TTCs in several cases and, as they suggest, the wall-clock time to insert new contacts also improves as τ increases.

For future investigations, we conjecture that our compact data structure can be simplified further so that the content of both its bit-vectors are merged into a single data structure. Our current insertion algorithm duplicates most operations in order to update both bit-vectors. Furthermore, each of these operations traverse a tree-like data structure from top to bottom. With a single tree-like data structure, our insertion algorithm could halve the number of traversals and, maybe, benefit from a better spatial locality. In another direction, our algorithm for $\text{INSERT}((A, D), t_1, t_2)$ only has time complexity $O(\log \tau)$ when both A and D encode leaves sparsely. Perhaps, a dynamic bit-vector data structure that holds a mix of leaves represented densely or sparsely can be employed to retain the $O(\log \tau)$ complexity while improving the overall runtime for other operations. Lastly, we expect soon to evaluate our new compact data structure on larger datasets and under other scenarios; for instance, in very sparse and real temporal graphs.

A Dynamic Disk-Based Data Structure for Temporal Reachability with Unsorted Contact Insertions

In Chapter 4, we introduced a data structure that supports the operations `add_contact(u, v, t)`, `can_reach(u, v, t_1, t_2)`, `is_connected(t_1, t_2)`, and `reconstruct_journey(u, v, t_1, t_2)`, in worst-case time $O(n^2 \log \tau)$, $O(\log \tau)$, $O(n^2 \log \tau)$, and $O(k \log \tau)$, respectively, where k is the length of the resulting journey, while using $O(n^2 \tau)$ space. The update algorithm maintains a Timed Transitive Closure (TTC), a concept that generalizes the transitive closure for temporal graphs based on *reachability tuples* (R-tuples), in the form (u, v, t^-, t^+) , representing journeys from vertex u to v departing at t^- and arriving at t^+ . This approach keeps the data structure in primary memory and the cost of maintaining large TTCs is prohibitive. In Chapter 5, we proposed a compact representation to alleviate this problem. However, even using our compact data structure, primary memory might not be sufficient.

We conducted a simple experiment to show how much space is necessary for maintaining temporal reachability. First, we generated random temporal graphs using the Edge-Markovian Evolving Graph (EMEG) model (CLEMENTI et al., 2008). In this model, if an edge is active (resp. not active) at time $t - 1$, then it has a probability p of disappearing (resp. probability q of appearing) at time t . We represented temporal graphs in memory using adjacency matrices storing, in each cell, timestamps at which edges are active. Then, we built the corresponding TTCs using the approach described in Chapter 4. In this experiment, we varied the number of vertices n and the number of time instances τ while fixing $p = 0.1$ and $q = 0.3$.

In Table 2, we see, for example, that a temporal graph with 512 vertices and $\tau = 64$ produced by the EMEG model has 2.8 million contacts, and we need around 33 MB of space to store it in memory. Besides, we need around 156 MB of space to store

Table 2 – Space for storing temporal graphs with n vertices, τ time instances and $|C|$ contacts, and their corresponding TTCs. Columns $data(\mathcal{G})$ and $data(\text{TTC})$ represent, respectively, the space in megabytes of the generated temporal graphs and their TTCs.

n	τ	$ C $	$data(\mathcal{G})$	$data(\text{TTC})$
32	8	1268	0.02	0.01
32	16	2670	0.03	0.12
32	32	5249	0.06	0.24
64	8	5539	0.08	0.27
64	16	10908	0.14	0.54
64	32	21421	0.26	1.08
64	64	42671	0.50	2.17
128	8	21203	0.31	1.13
128	16	43011	0.55	2.30
128	32	86746	1.06	4.63
128	64	173479	2.05	9.31
256	8	86574	1.24	4.67
256	16	174970	2.25	9.51
256	32	346994	4.22	19.17
256	64	696436	8.22	38.54
512	8	349114	5.00	19.01
512	16	702294	9.04	38.66
512	32	1396033	16.98	78.00
512	64	2800520	33.05	156.64

the corresponding TTC, which, in this case, it is almost five times the space needed to store the temporal graph.

Then, we built a linear regression model with the data presented in Table 2 in order to extrapolate the input parameters. Consider, for example, the scenario in which one million people use a bluetooth device that registers when and who gets close to each other and sends this information to a centralized server. Consider also that each individual makes 30 contacts per day on average. In this setting, by using our model, we could check that a centralized server would require at least 100 GB of space in less than a year to store just the plain contacts as a temporal graph. If one needs to support reachability queries by using a TTC, it would be necessary roughly 600 GB of space.

Motivated by such scenarios, we investigate the problem of maintaining TTCs on disk. A simple, but not efficient, approach would be to adapt our previous approach. Briefly, our previous approach maintains self-balanced Binary Search Trees (BSTs) containing time intervals for each pair of vertices in order to retrieve reachability information. However, this approach does not consider data locality, thus each update operation would randomly access an excessive amount of pages on disk to retrieve information from each

BST. For instance, if we use B^+ -trees (ABEL, 1984) as a replacement for BSTs, the algorithm for `add_contact`(u, v, t) would access $O(n^2)$ B^+ -trees and, in each B^+ -tree, it would access $O(\log_B \tau)$ pages, where B is the page size, resulting in $O(n^2 \log_B \tau)$ random accesses on disk.

In this chapter, we propose an incremental disk-based data structure that reduces the number of disk accesses for both update and query operations while prioritizing sequential accesses. The core idea of our novel approach is to explicitly maintain an *expanded* set of non-redundant R-tuples containing $n^2\tau$ elements. Conceptually, we maintain it using two 3-dimensional arrays, M_{out} and M_{in} , of size $n \times \tau \times n$ such that $M_{out}[u, t^-, v] = t^+$ and $M_{in}[v, t^+, u] = t^-$. The former supports querying the earliest arrival t^+ a journey departing from vertex u at time t^- can arrive at vertex v , and the latter supports querying the latest departure t^- a journey arriving to vertex v at time t^+ can depart from vertex u .

Our algorithm to compute `add_contact`(u, v, t) eagerly updates both arrays accessing $O(n^2\tau/B)$ disk pages in the worst case. Despite having a linear factor on τ instead of logarithmic, the expected cost of our update routine reduces considerably as we insert new contacts. This is because journey schedules become stricter and the probability of replacing them with faster ones reduces. Since we explicitly maintain reachability information, our algorithms to answer `can_reach`(u, v, t_1, t_2), `is_connected`(t_1, t_2), and `reconstruct_journey`(u, v, t_1, t_2) access, respectively, $\Theta(1)$, $\Theta(n^2/B)$ and $\Theta(n/B)$ pages sequentially.

We compare our novel data structure with a naïve adaptation of our previous approach using B^+ -trees as replacement for BSTs. Our experiments show that our novel data structure performs better on the synthetic datasets and on the majority of real-world datasets we used. Even though the worst-case complexity of our algorithm for the `add_contact`(u, v, t) operation is linear in τ instead of logarithmic, it runs much faster on average. We attribute this behavior to the fact that as new contacts are inserted, our data structure updates on average only a few cells of both arrays M_{out} and M_{in} .

The content present in this chapter was published on the arXiv repository (BRITO; ALBERTINI; TRAVENÇOLO, 2023) available at <https://arxiv.org/abs/2306.13937>.

We organized this chapter as follows. In Section 6.1, we define our expanded set of R-tuples, introduce our new data structure to represent TTCs on disk, and provide low-level primitives for manipulating them. In Section 6.2, we describe our algorithms for each operation using our new data structure along with their complexities in terms of the number of disk accesses. In Section 6.3, we investigate the execution of our algorithms by comparing them with our implementation using B^+ -trees. Finally, Section 6.4 concludes with some remarks and open questions.

6.1 Disk-Based Timed Transitive Closure

In this section, we describe our novel approach to maintain TTCs in secondary memory. First, in Section 6.1.1, we define the concept of an *expanded* set of representative R-tuples and show that it has size $\Theta(n^2\tau)$. Then, in Section 6.1.2, we introduce our new data structure that uses this expanded set in order to improve the maintenance of data in non-uniform access storages and provide direct access to reachability information.

6.1.1 Expanded Reachability Tuples (Expanded r-tuples)

The data structure introduced in Chapter 4 spreads the minimal set of R-tuples into multiple BSTs, each one concerning a unique pair of vertices. Previously, we stored these BSTs in separated regions of memory and, therefore, the organization of data is not optimal when working with storages that have non-uniform access time.

In order to mitigate this problem, we define an expanded set of R-tuples (u, v, t^-, t^+) that is easier to maintain sequentially, since we can use continuous arrays indexed by t^- or t^+ . First, we define the *left* and *right* expansion of a single R-tuple.

Definition 11 (Left and right expansion). *The left expansion of an R-tuple $r = (u, v, t^-, t^+)$ is the set containing all R-tuples (u, v, t, t^+) for $1 \leq t \leq t^-$. Similarly, the right expansion of r is the set containing all R-tuples (u, v, t^-, t) for $t^+ \leq t \leq \tau + \delta$.*

The R-tuples produced by the left expansion of an R-tuple r are valid because a source vertex departing earlier can simply wait until the departure of r , and take the original journey described by r . Similarly, the R-tuples produced by the right expansion of r are also valid because, after taking the original journey described by r , a destination vertex can simply wait until the arrival of the new R-tuple.

Applying both expansions to every R-tuple of a minimal set \mathcal{R} and taking the union of the sets produced by the same expansion creates two separated expanded sets, the *left-expanded* set $\mathcal{R}_{\text{left}}$, and the *right-expanded* set $\mathcal{R}_{\text{right}}$. For each expanded set, we define an inclusion operator.

Definition 12 (Left and right inclusion). *Given any two R-tuples $r_1 = (u_1, v_1, t_1^-, t_1^+)$ and $r_2 = (u_2, v_2, t_2^-, t_2^+)$ in $\mathcal{R}_{\text{left}}$, $r_1 \subseteq_{\text{left}} r_2$ if and only if $u_1 = u_2$, $v_1 = v_2$, $t_1^- = t_2^-$, and $t_1^+ \leq t_2^+$. Similarly, if r_1 and r_2 are in $\mathcal{R}_{\text{right}}$, $r_1 \subseteq_{\text{right}} r_2$ if and only if $u_1 = u_2$, $v_1 = v_2$, $t_1^- \geq t_2^-$, and $t_1^+ = t_2^+$.*

However, R-tuples produced by expansion can share redundant information. For example, consider the R-tuples $r_1 = (a, b, 2, 7)$ and $r_2 = (a, b, 2, 9)$. Both R-tuples represent journeys that depart from vertex a at time 2 and arrives at vertex b , one at time 7 and the other at time 9. In this case, r_2 can be safely discarded since we can take a

journey represented by r_1 ending at time 7 and wait at vertex v until time 9. Redundancy of R-tuples in $\mathcal{R}_{\text{left}}$ and $\mathcal{R}_{\text{right}}$ is treated differently using their corresponding inclusion operators.

Definition 13 (Left and right redundancy). *Let $r \in \mathcal{R}_{\text{left}}$, r is called left-redundant in $\mathcal{R}_{\text{left}}$ if there is $r' \in \mathcal{R}_{\text{left}}$ such that $r' \subseteq_{\text{left}} r$. Similarly, if $r \in \mathcal{R}_{\text{right}}$, r is called right-redundant in $\mathcal{R}_{\text{right}}$ if there is $r' \in \mathcal{R}_{\text{right}}$ such that $r' \subseteq_{\text{right}} r$. A set $\mathcal{R}_{\text{left}}^*$ with no left-redundant R-tuple is called left not-redundant and a set $\mathcal{R}_{\text{right}}^*$ with no right-redundant R-tuple is called right not-redundant.*

Lemma 12. *The maximum size of a left not-redundant or right not-redundant set of R-tuples for \mathcal{G} is $\Theta(n^2\tau)$.*

Proof. We need only to prove the upper bound case since an *unexpanded* not-redundant set already has $\Theta(n^2\tau)$ R-tuples, see Chapter 4. There are $\Theta(n^2)$ ordered pairs of vertices. Thus, it suffices to show that for each pair (u, v) , the number of incomparable R-tuples whose starting vertex is u and whose ending vertex is v is $\Theta(\tau)$. Let S_l and S_r be, respectively, left not-redundant and a right not-redundant sets of such tuples. There can only exist one R-tuple for a departure $t^- \in [1, \tau]$ in S_l (Definition 12), otherwise S_l would be redundant, which implies $|S_l| \leq \tau$. Similarly, There can only exist one R-tuple for an arrival $t^+ \in [1, \tau]$ in S_r , which also implies $|R_l| \leq \tau$. \square

6.1.2 Encoding the TTC on Disk

We encode the TTC using two 3-dimensional arrays, $M_{\text{out}}[u, t^-, v] = t^+$ and $M_{\text{in}}[v, t^+, u] = t^-$, both with dimensions $n \times \tau \times n$, representing expanded sets of R-tuples. Each cell in M_{out} represents a R-tuple in $\mathcal{R}_{\text{left}}^*$ by storing the earliest arrival t^+ at which a vertex u departing at time t^- can reach a vertex v through a journey. If there is a cell $M_{\text{out}}[u, t^-, v] = t^+$, then all cells $M_{\text{out}}[u, t, v]$, for $t \in [1, t^- - 1]$ must have an arrival $t^+ \leq t^+$, since a journey from u departing at a time $t < t^-$ can simply wait at vertex u until time t^- . Similarly, each cell in M_{in} represents an R-tuple in $\mathcal{R}_{\text{right}}^*$ by storing the latest departure t^- at which a vertex v can arrive at time t^+ to a vertex u through a journey. If there is a cell $M_{\text{in}}[v, t^+, u] = t^-$, then all cells $M_{\text{in}}[v, t, u]$, for $t \in [t^+ + 1, \tau + \delta]$, must have a departure $t^+ \geq t^+$, since a journey to v arriving at a time $t > t^+$ can just wait until time t^+ after arriving at v . The creation of a TTC initializes all cells of M_{out} to ∞ and all cells of M_{in} to $-\infty$. Figure 15 illustrates both M_{out} and M_{in} .

Internally, we represent M_{out} and M_{in} as one-dimensional arrays using, respectively, the mapping functions $F_{\text{out}}: (u, t^-, v) \mapsto n(u\tau + \tau - (t^- + 1)) + v + 1$ and $F_{\text{in}}: (v, t^+, u) \mapsto n(v\tau + t^+ - \delta) + u + 1$. Observing Figure 15, F_{out} arranges the cells of M_{out} by row (left to right) and, for each source vertex, later departures come first. F_{in} also arranges M_{in} by row but, in contrast, for each destination vertex, earlier arrivals

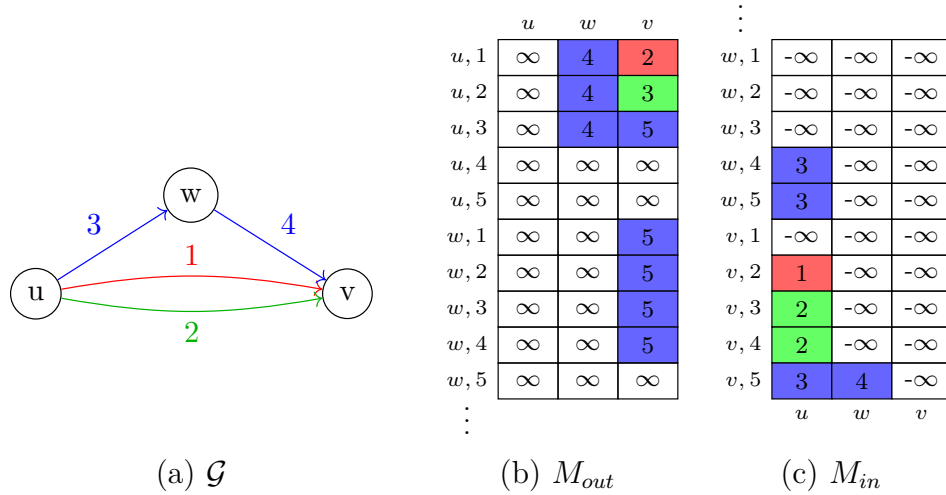


Figure 15 – Temporal graph and its associated reachability data structure. In (a), we show a temporal graph with three vertices. Numbers on edges represent the time in which edges are active. Edges with the same color form a journey from vertex u to vertex v . In (b), we show the corresponding arrays M_{out} and M_{in} considering $\delta = 1$. Both arrays are depicted as 2-dimensional arrays by grouping their first two dimensions. For instance, $M_{out}[u, 2, w] = M_{out}[(u, 2), w] = 3$. Cells have the same color as the contacts, *i.e.*, the edge at a timestamp, that originated the update. M_{out} stores the minimum arrivals to destinations and M_{in} stores the maximum departures from origins.

come first. By subtracting δ from t^+ in F_{in} , we ensure all t^+ values fit in M_{in} . Thus, reading sequentially the range $[F_{out}(u, t^-, 1), F_{out}(u, t^-, n)]$ from M_{out} gives direct access to the earliest arrivals to reach all vertices when departing from u at time t^- . Similarly, reading sequentially the range $[F_{in}(v, t^+, 1), F_{in}(v, t^+, n)]$ from M_{in} gives direct access to the latest departures to leave all vertices when arriving at v at time t^+ .

Finally, assuming a general function F that maps to F_{out} , whether accessing M_{out} , or F_{in} , whether accessing M_{in} , we provide the following low-level operations for manipulating our data structures on disk:

1. $READ_CELL(M, w_1, t, w_2)$, which returns the value of M (M_{out} or M_{in}) at position $F(w_1, t, w_2)$;
2. $READ_ADJACENCY(M, w, t)$, which returns a list containing the values of M in the interval $[F(w, t, 1), F(w, t, n)]$, *i.e.*, the minimum timestamps to arrive at any vertex while departing from w at time t ;
3. $WRITE_ADJACENCY(M, w, t, L)$, which replaces the values of M values in the interval $[F(w, t, 1), F(w, t, n)]$ with the values of the list L , *i.e.*, the maximum timestamps to depart from any vertex while arriving at w at time t .

Operation (1) accesses $O(1)$ pages on disk, while operations (2) and (3) access $O(n/B)$ pages, where B is the page size.

6.2 The Four Operations

In this section, we describe algorithms for the operations four operations described in Chapter 1: the update operation `add_contact`(u, v, t); the query operations `can_reach`(u, v, t_1, t_2) and `is_connected`(t_1, t_2); and the reconstruction operation `reconstruct_journey`(u, v, t_1, t_2). In Section 6.2.2, we present our algorithm for `add_contact`(u, v, t) that receives a contact and adds to our data structure the reachability information related to the new available journeys passing through it. In Section 6.2.1, we briefly describe algorithms for `can_reach`(u, v, t_1, t_2) and `is_connected`(t_1, t_2) since, as reachability information can be directly accessed, they are straightforward. Finally, in Section 6.2.3, we detail our algorithm for `reconstruct_journey`(u, v, t_1, t_2) that reconstructs a valid journey by concatenating one contact at a time.

6.2.1 Reachability and Connectivity Queries

Both algorithms for `can_reach`(u, v, t_1, t_2) and `is_connected`(t_1, t_2) are straightforward. The algorithm to perform `can_reach`(u, v, t_1, t_2) comprises testing whether $\text{READ_CELL}(M_{out}, u, t_1, v) \leq t_2$ while accessing only a single page from disk. The algorithm to perform `is_connected`(t_1, t_2), for each origin vertex $u \in V$, calls $tmp \leftarrow \text{READ_ADJACENCY}(M_{out}, u, t_1)$ and then for each destination vertex $v \in V$, it checks whether $tmp[v] \leq t_2$. As soon as a check is negative, the algorithm returns false; otherwise, it returns true. Therefore, the algorithm sequentially accesses $O(n^2/B)$ pages on disk.

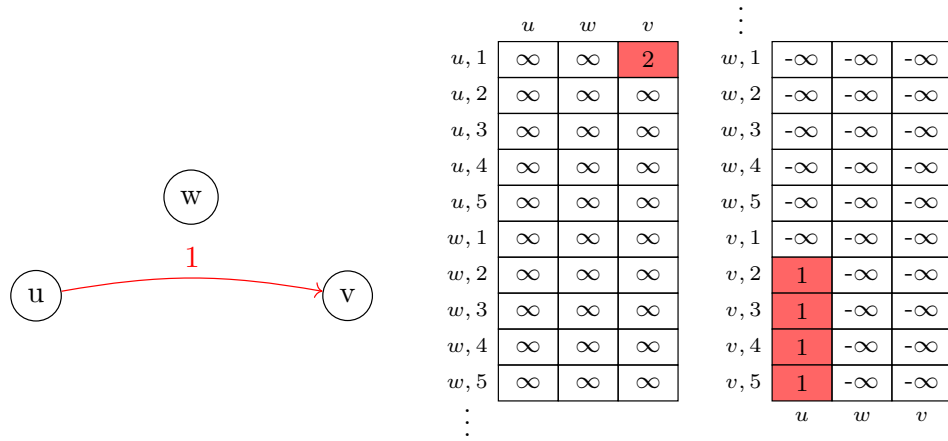
6.2.2 Update Operation

An algorithm to perform `add_contact`(u, v, t) must first add the reachability information regarding the new trivial journey \mathcal{J}_{triv} from vertex u to vertex v departing at time t and arriving at time $t + \delta$. Next, for all vertices w^- that can reach u when arriving at a timestamp earlier than or exactly t , the algorithm updates the reachability information from w^- to v whether the new available journey passing through \mathcal{J}_{triv} has later departure. Then, for all vertices w^+ that v can reach when departing at a timestamp later than or exactly $t + \delta$, the algorithm updates the reachability information from u to w^+ whether the new available journey passing through \mathcal{J}_{triv} has earlier arrival. Finally, the algorithm must consider all new available journeys from vertices w^- to vertices w^+ that pass through \mathcal{J}_{triv} and update the current reachability information if necessary.

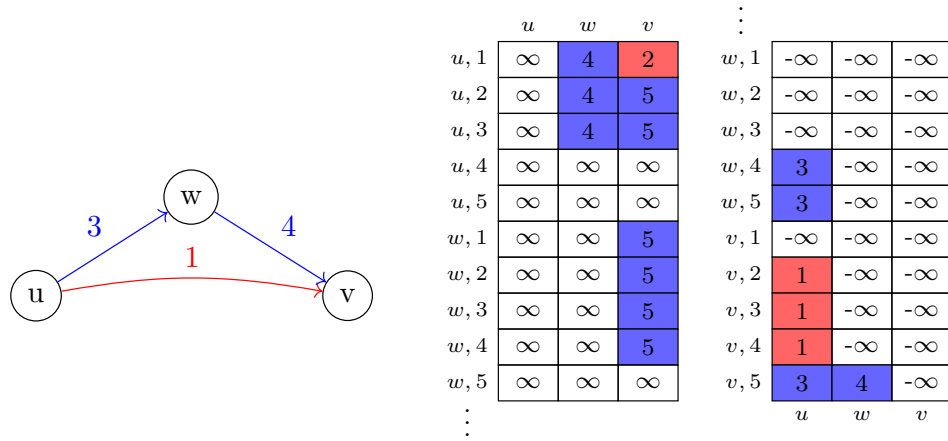
Algorithm 8 describes the maintenance of both arrays M_{out} and M_{in} when inserting a new contact. At line 1, the algorithm checks if the structure already has the information of the new contact (u, v, t) . If it still has not, at line 2, it retrieves the latest departures of journeys departing from vertices w^- and arriving at vertex u at time t as an array T^- . At line 3, the algorithm retrieves the earliest arrivals of journeys departing from vertex v at time $t + \delta$ and arriving at vertices w^+ as an array T^+ . At lines 4 and 5, it sets the reachability information about the new trivial journey $\mathcal{J}_{triv} = (u, v, t)$ that departs at time t and arrives at $t + \delta$. From line 6 to 14, the algorithm eagerly updates all cells $M_{out}[w^-, t', w^+] = t^+$ for $t^- \geq t' \geq 1$. In this part, the algorithm proceeds by first iterating through all vertices w^- , *i.e.*, those that reached u before than or exactly at time t , and retrieving their departures t^- . Then, it progressively retrieves the current arrivals to reach vertices w^+ when departing at time t' , by reading the range $[F_{out}(w^-, t', 1), F_{out}(w^-, t', n)]$, and updates it whether the new journeys passing through \mathcal{J}_{triv} have earlier arrivals. Note that vertices that could not reach u before than or exactly at time t have their arrival equals to $-\infty$; therefore, they are not considered in the while loop starting at line 8. This process continues until the current reachability information in the entire range does not change or $t' < 1$. Similarly, from line 15 to 23, the algorithm eagerly updates all cells $M_{in}[w^+, t'', w^-] = t^-$ for $t^+ \leq t'' \leq \tau + \delta$. The algorithm proceeds by first iterating through all vertices w^+ , *i.e.*, those that v can reach departing after or exactly at time $t + \delta$, and retrieving their arrivals t^+ . Then, it progressively retrieves the current departures in which vertices w^- departs present in range $[F_{in}(w^+, t'', 1), F_{in}(w^+, t'', n)]$ and updates it whether the new available journeys passing through \mathcal{J}_{triv} have later departures. This process continues until the current reachability information in the range does not change or $t'' > \tau + \delta$. Figure 16 illustrates the addition of new contacts to a temporal graph along with the maintenance of the arrays M_{out} and M_{in} .

Theorem 13. *Algorithm 8 access $O(n^2\tau/B)$ pages on disk.*

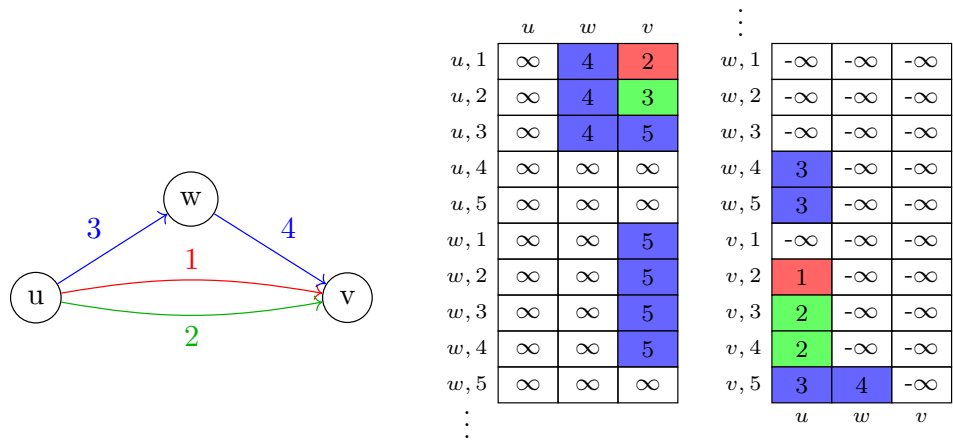
Proof. The READ_CELL operation in line 1 access a single page. The two READ_ADJACENCY operations in lines 2 and 3 access $O(n/B)$ sequential pages each. At lines 4 and 5, the algorithm writes the reachability of the new trivial journey $\mathcal{J}_{triv} = \{(u, v, t)\}$ in primary memory. The for loop starting at line 6 iterates over n vertices w^- and, the while loop starting at line 8 iterates through $O(\tau)$ timestamps t' . In each of the $O(n\tau)$ iterations, it calls READ_ADJACENCY in order to read n cells, and then (possibly) calls WRITE_ADJACENCY to write the n cells back while accessing, in each operation, $O(n/B)$ sequential pages. Due to our mapping function F_{out} , at every timestamp t' , the algorithm will read a page that is arranged sequentially on disk. The loop from line 15 to 23 does a similar computation. \square



(a) added contact $(u, v, 1)$



(b) added contacts $(u, w, 3)$ and $(w, v, 4)$



(c) added contact $(u, v, 2)$

Figure 16 – Maintenance of our disk-based TTC, encoded as two arrays, M_{out} (left table) and M_{in} (right table), in different scenarios. Both arrays are depicted as 2-dimensional arrays by grouping their first two dimensions. In (a), the contact $(u, v, 1)$ is inserted in the temporal graph and thus M_{out} and M_{in} are updated using the information present in the left and right expansions of the R-tuple $(u, v, 1, 2)$. In (b), both contacts $(u, w, 3)$ and $(w, v, 4)$ are inserted and a non-trivial journey from u to v becomes possible. Finally, in (c), the contact $(u, v, 2)$ is inserted, allowing a faster journey departing from u at time 2 and triggering the update of some cells of M_{out} and M_{in} .

Algorithm 8 `add_contact`(u, v, t)**Require:** $u, v \in V$ with $u \neq v$, $n = |V|$, $t \in \mathcal{T}$, $\tau, \delta, M_{out}, M_{in}$

```

1: if READ_CELL( $M_{out}, u, t, v$ )  $\neq t + \delta$  then            $\triangleright$  check whether  $(u, v, t)$  was inserted
2:    $T^- \leftarrow$  READ_ADJACENCY( $M_{in}, u, t$ )
3:    $T^+ \leftarrow$  READ_ADJACENCY( $M_{out}, v, t + \delta$ )
4:    $T^-[u] \leftarrow t$                                       $\triangleright$  add the new trivial journey information
5:    $T^+[v] \leftarrow t + \delta$ 
6:   for  $w^-$  from 1 up to  $n$  do                              $\triangleright$  update  $M_{out}$  with new journeys from  $w^-$ 
7:      $t' \leftarrow T^-[w^-]$ 
8:     while  $t' \neq -\infty$  and  $t' \geq 1$  do                  $\triangleright$  loop for  $t^- \geq t' \geq 1$ 
9:        $T_{cur}^+ \leftarrow$  READ_ADJACENCY( $M_{out}, w^-, t'$ )
10:       $T_{cur}^+[w^+] \leftarrow \min(T_{cur}^+[w^+], T^+[w^+])$  for  $w^+ \in [1, n]$ 
11:      if  $T_{cur}^+$  has not changed then
12:        break
13:      WRITE_ADJACENCY( $M_{out}, w^-, t', T_{cur}^+$ )
14:       $t' \leftarrow t' - 1$ 
15:   for  $w^+$  from 1 up to  $n$  do                              $\triangleright$  update  $M_{in}$  with new journeys to  $w^+$ 
16:      $t'' \leftarrow T^+[w^+]$ 
17:     while  $t'' \neq \infty$  and  $t'' \leq \tau + \delta$  do        $\triangleright$  loop for  $t^+ \leq t'' \leq \tau + \delta$ 
18:        $T_{cur}^- \leftarrow$  READ_ADJACENCY( $M_{in}, w^+, t''$ )
19:        $T_{cur}^-[w^-] \leftarrow \max(T_{cur}^-[w^-], T^-[w^-])$  for  $w^- \in [1, n]$ 
20:       if  $T_{cur}^-$  has not changed then
21:         break
22:       WRITE_ADJACENCY( $M_{in}, w^+, t'', T_{cur}^-$ )
23:      $t'' \leftarrow t'' + 1$ 

```

6.2.3 Journey Reconstruction

For the `reconstruct_journey`(u, v, t_1, t_2) query, we need to augment each cell of M_{in} with the first successor vertex of the corresponding journeys. Algorithm 8 can be trivially modified to include this information. For instance, the successor vertex of a trivial journey from a contact (u, v, t) is the vertex v since it is the first successor of u . Furthermore, the algorithm would also need to consider successor vertices when composing new R-tuples to update M_{out} and M_{in} .

Algorithm 9 gives the details to process the `reconstruct_journey`(u, v, t_1, t_2) query. Its goal is to reconstruct a journey by unfolding the intervals and successor fields. At line 1, it initializes an empty journey \mathcal{J} . At line 2, it retrieves the earliest time t^+ a journey from vertex u departing at time t_1 can arrive at vertex v by reading on disk the entry $M_{out}[u, t_1, v]$. If $t^+ \leq t_2$, it reconstructs the resulting journey, otherwise; it returns an empty journey since there is no journey completely in the interval $[t_1, t_2]$. From lines 4 to 10, it reconstructs the resulting journey by: first, at lines 4 and 5, initializing the successor vertex $succ$ to u , and accessing on disk all the entries $M_{in}[v, t^+, w]$ for $w \in V$; then, from lines 6 to 10, the journey is reconstructed by iteratively accessing the next earliest departing time t^- and the corresponding successor vertex $next_succ$ that reaches

v at time t^+ while concatenating the contact $(succ, next_succ, t^-)$ at the end of \mathcal{J} and updating the current successor vertex.

Algorithm 9 `reconstruct_journey`(u, v, t_1, t_2)

Require: $[t_1, t_2] \subset \mathcal{T}, u, v \in V$ with $u \neq v$

```

1:  $\mathcal{J} \leftarrow \{\}$ 
2:  $(t^+, \_)\leftarrow \text{READ\_CELL}(M_{out}, u, t_1, v)$ 
3: if  $t^+ \leq t_2$  then
4:    $succ \leftarrow u$ 
5:    $in \leftarrow \text{READ\_ADJACENCY}(M_{in}, v, t^+)$ 
6:   while  $succ \neq v$  do
7:      $t^- \leftarrow in[succ].t$ 
8:      $next\_succ \leftarrow in[succ].succ$ 
9:      $\mathcal{J} \leftarrow \mathcal{J} \cup (succ, next\_succ, t^-)$ 
10:     $succ \leftarrow next\_succ$ 
11: return  $\mathcal{J}$ 

```

Theorem 14. *Algorithm 9 sequentially accesses $O(n/B)$ pages on disk, where n is the number of vertices and B is the page size.*

Proof. The algorithm accesses one page by calling `READ_CELL`(M_{out}, u, t_1, v) at line 2. After that, it is known whether a journey exists. If a journey exists, it sequentially accesses n/B pages by calling `READ_ADJACENCY`(M_{in}, v, t^+) at line 5. The result has all information needed to reconstruct a valid journey. Finally, in the loop from line 6 to line 8, the algorithm extends the resulting journey by one contact at each iteration using information already in memory. Thus, the number of pages accessed is dominated by the call `READ_ADJACENCY`(M_{in}, v, t^+). \square

6.3 Experiments

In this section, we present experiments comparing our novel data structure based on sequential arrays with the approach we adapted from our previous data structure described in Chapter 4 using B^+ -trees as a replacement for self-balanced Binary Search Trees (BSTs). Briefly, our adaptation of our previous data structure stores, in a matrix $n \times n$, pointers to BSTs containing time intervals. In each BST, only non-redundant intervals are kept, *i.e.*, those that do not contain another interval in the same tree. We used *join-based* operations in order to remove sequences of non-redundant intervals in $\log \tau$ time. These operations for B^+ -trees can be found in Appendix B.

In the following, we present two experiments in Sections 6.3.1 and 6.3.2. In the first one, we inserted unsorted contacts from complete temporal graphs, incrementally, in both data structures using the operation `add_contact`(u, v, t). In the second one, we inserted shuffled contacts from real-world datasets.

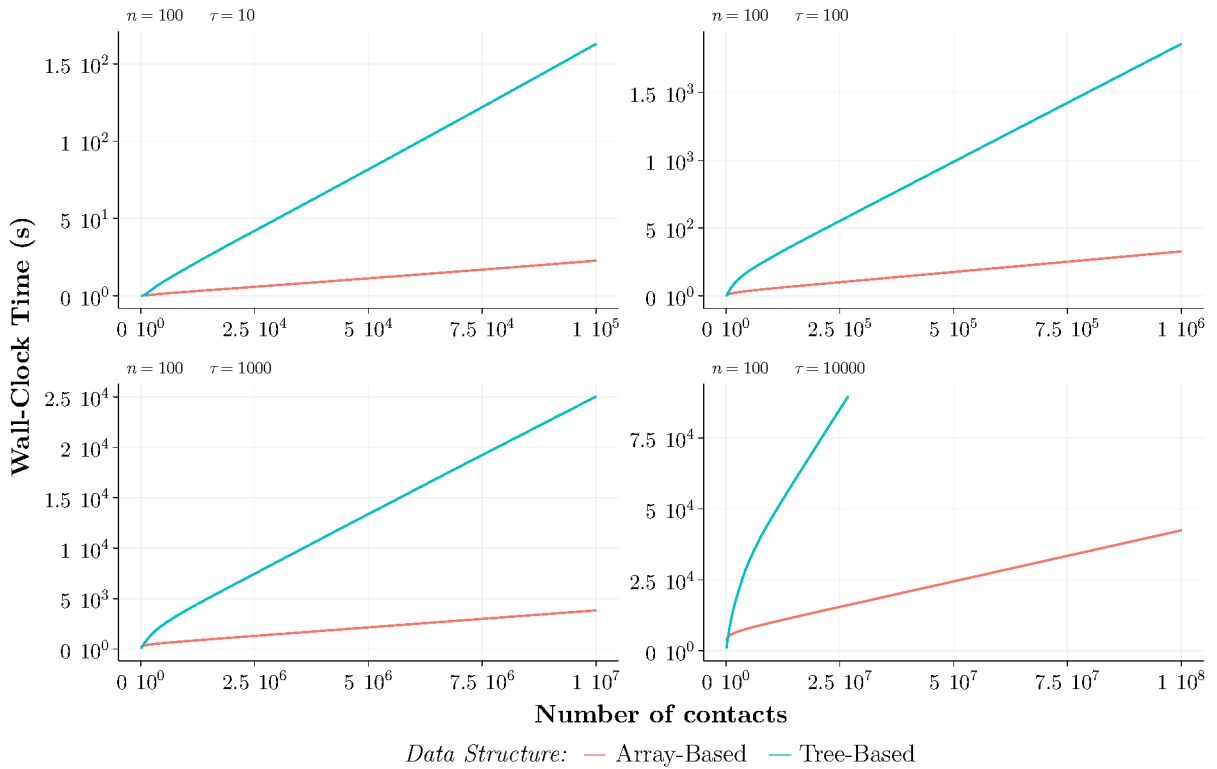


Figure 17 – Cumulative wall-clock time to maintain data structures for reachability queries on synthetic data. We inserted shuffled contacts from complete temporal graphs into the data structures varying the number of timestamps τ while fixing the number of vertices to 100. Red lines represent our novel data structure based on sequential arrays. Blue lines represent our adaptation of the approach introduced in Chapter 4 using B^+ -trees as self-balanced BSTs.

6.3.1 Experiments with Synthetic Data

In this first experiment, we generated complete temporal graphs with the number of vertices fixed to 100 and varied the number of timestamps τ from 10 to 10000. Then, we inserted their shuffled contacts in both data structures using the `add_contact(u, v, t)` operation. The time to pre-allocate and initialize the arrays on disk for our array-based data structure was not considered in the total time. We note that this extra cost can be high for large parameters; therefore, one should consider it whenever applicable.

Figure 17 shows the mean cumulative wall-clock time, averaged over 10 executions, to maintain both data structures as new unsorted contacts were inserted. We see that our novel data structure performs better for all configurations. Even though the worst-case complexity of our algorithm for the `add_contact(u, v, t)` operation is linear in τ instead of logarithmic, it runs much faster using synthetic data. We attribute this behavior to the fact that as new contacts are inserted, the probability of composing better R-tuples, *i.e.* journeys, decreases rapidly and, thus, our data structure updates on average only a few cells per contact insertion.

Next, we argue why the running time of our algorithm reduces with the addition of contacts. Each pair of vertices (u, v) are associated to a set \mathcal{I} containing intervals $[t^-, t^+] \subseteq [1, \tau]$ in which u can reach v departing at t^- and arriving at t^+ . For a particular pair of vertices, when an algorithm inserts a new interval I , all intervals I' such that $I \subseteq I'$ can be safely removed since they become redundant. Our data structure organizes these intervals in the arrays M_{out} and M_{in} , which fix, respectively, the left and right endpoints, and our update algorithm discards redundant intervals by updating their cells accordingly by using Definition 13.

Consider the hierarchy of intervals illustrated in Figure 18(a) for $\tau = 4$. Each interval with length l is linked to the intervals with length $l - 1$ that it totally encloses. For example, interval $[0, 5]$, with length 5, links to intervals $[0, 4]$ and $[1, 5]$, with length 4, because $[0, 4] \subseteq [0, 5]$ and $[1, 5] \subseteq [0, 5]$. Initially, all intervals are available for insertion in our data structure. When a new interval $[1, 2]$ is inserted, as shown in Figure 18(b), all intervals that contain it, including itself, are not available for insertion anymore.

Our update algorithm conceptually removes these intervals by drawing left and right frontiers separating available and non-available intervals starting from $[1, 2]$. For instance, intervals $[1, 2]$ and $[0, 2]$, which belong to the left frontier, are updated in M_{in} since they share the same right endpoint, and intervals $[1, 2]$, $[1, 3]$, $[1, 4]$ and $[1, 5]$, which belong to the right frontier, are updated in M_{out} since they share the same left endpoint. In this process, up to τ cells are updated in both M_{in} and M_{out} .

Next, when a new interval $[3, 5]$ is inserted, as shown in Figure 18(c), our algorithm must, again, draw the left and right frontiers starting from $[3, 5]$; however, it does not need to advance previously drawn frontiers. Here, only intervals $[3, 5]$ and $[2, 5]$, which belong to the left frontier, are updated in M_{in} . In Figure 18(d), interval $[2, 3]$ is inserted and the same process repeats. We see that as new intervals are inserted, the number of available intervals rapidly reduces. Thus, even though our algorithm has complexity $O(n^2\tau/B)$, it can run much faster when considering a sequence of contact insertions since the number of cells to be updated reduces.

Moreover, it is guaranteed that, for each new contact (u, v, t) , our algorithm will make unavailable for insertion every interval that is still available inside and at the frontiers starting from $[t, t + \delta]$ in the lowest level of the hierarchy associated with (u, v) .

6.3.2 Experiments with Real-World Datasets

In this second experiment, we downloaded small and medium real-world available on <https://networkrepository.com/dynamic.php>, and pre-processed them using our script available on <https://bitbucket.com/luizufu/temporalgraph-datasets-preprocessing>. During the preprocessing, we relabeled the

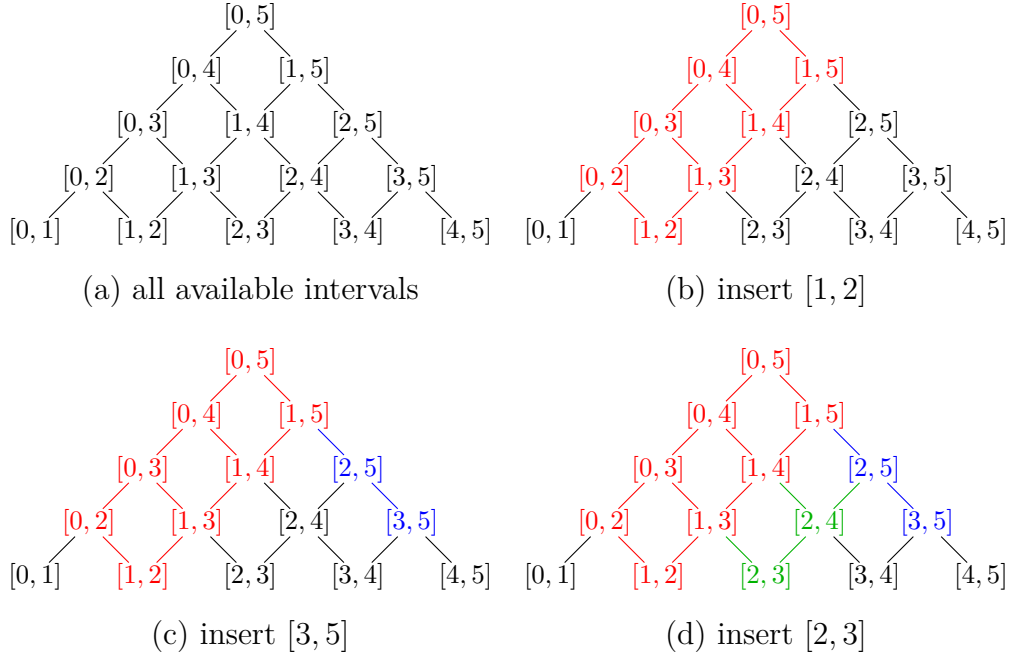


Figure 18 – Illustration of the process performed by our update algorithm considering a fixed pair of vertices (u, v) from a temporal graph with $\tau = 4$. Available intervals for insertion are colored in black, and invalidated intervals, *i.e.* intervals that should not be considered anymore by our update algorithm, are colored in different colors. Links represent the direct containment relation between intervals with length l and intervals with length $l - 1$.

vertices and shifted the timestamps of each dataset so that vertex identifiers were between $[1, n]$ and timestamp values start from 1. Then, we inserted the shuffled contacts of each dataset in both data structures using the `add_contact` (u, v, t) operation. We assumed that all used datasets represent temporal digraphs, and we used $\delta = 1$, *i.e.*, traversing any contact takes one time unit.

Table 17 shows the mean wall-clock time, averaged over 10 executions, to insert all shuffled contacts of each dataset into both data structures. We see that our novel data structure performs better on most of datasets. However, for the largest datasets, `copresence-LH10` and `copresence-LyonSchool`, the tree-based data structure performed better. Both datasets have high values for τ and low density. It means that, as density is too small, each insertion of a contact (u, v, t) may trigger an initial update over arrays M_{out} and M_{in} that will touch many cells on disk. As in Figure 18(b), for most insertions, our update algorithm will draw left and right frontiers on the almost empty hierarchy associated with the pair of vertices (u, v) starting from interval $[t, t + \delta]$. Therefore, in this case, the linear factor on τ from the cost $O(n^2\tau/B)$ of our update algorithm will have a bigger impact on the running time since the sequence of insertions is not sufficiently long for our algorithm to benefit from later insertions.

Table 3 – Total wall-clock time in seconds to insert all shuffled contacts from real-world datasets with number of vertices n , number of timestamps τ , number of contacts into data structures for reachability queries, and the density of the temporal graph represented by the dataset. Values were rounded to two decimal places. Array-based refers to our novel data structure and tree-based refers to our implementation of the approach introduced in (BRITO et al., 2022) using B⁺-trees as BSTs replacement. Executions that reached the time limit of 5 hours are marked with the symbol “-”.

dataset	n	τ	contacts	density	Array-Based	Tree-Based
aves-sparrow	52	2	516	0.1	0.01 ± 0.00	0.07 ± 0.00
aves-weaver	445	23	1423	0.003	0.19 ± 0.00	1.16 ± 0.01
aves-wildbird	202	6	11900	0.05	0.97 ± 0.01	9.52 ± 0.15
ant-colony1	113	41	111578	0.46	25.84 ± 0.19	161.3 ± 1.03
ant-colony2	131	41	139925	0.2	43.96 ± 0.49	261.98 ± 1.96
ant-colony3	160	41	241280	0.23	89.37 ± 0.73	524.79 ± 6.71
ant-colony4	102	41	81599	0.19	16.49 ± 0.12	104.62 ± 1.27
ant-colony5	152	41	194317	0.21	166.93 ± 3.63	526.03 ± 144.32
ant-colony6	164	39	247214	0.24	88.99 ± 0.93	608.87 ± 167.97
copresence-LH10	73	259181	150126	0.0001	-	61.5 ± 0.53
copresence-LyonSchool	242	117721	6594492	0.001	-	14887.45 ± 1576.49
kilifi-within-households	54	59	32426	0.19	0.09 ± 0.00	0.21 ± 0.00
mammalia-primate	25	19	1340	0.12	0.09 ± 0.00	0.33 ± 0.01
mammalia-raccoon	24	52	1997	0.06	0.21 ± 0.00	0.44 ± 0.01
mammalia-voles-bhp	1686	63	5324	0.00003	13.76 ± 0.82	19.22 ± 0.24
mammalia-voles-kcs	1218	64	4258	0.00004	7.92 ± 0.27	11.78 ± 0.09
mammalia-voles-plj	1263	64	3863	0.00003	6.18 ± 0.23	10.68 ± 0.04
mammalia-voles-rob	1480	63	4569	0.00003	10.28 ± 0.41	15.09 ± 0.12
tortoise-bsv	136	4	554	0.008	0.01 ± 0.00	0.14 ± 0.01
tortoise-cs	73	10	258	0.005	0.01 ± 0.00	0.05 ± 0.00
tortoise-fi	787	9	1713	0.0003	0.15 ± 0.00	2.71 ± 0.01
trophallaxis-colony1	41	8	308	0.02	0.02 ± 0.00	0.06 ± 0.00
trophallaxis-colony2	39	8	330	0.03	0.02 ± 0.00	0.05 ± 0.00

6.4 Concluding remarks

We presented in this chapter an incremental disk-based data structure to solve the dynamic connectivity problem in temporal graphs. Our data structure prioritizes query time, answering reachability queries by accessing only one page. Based on the ability to retrieve quickly the reachability information among vertices inside time intervals, it can: insert contacts in a non-chronological order accessing $O(n^2\tau/B)$ pages, where B is the size of disk pages; check whether a temporal graph is connected within a time interval accessing $O(n^2/B)$ pages, and reconstruct journeys accessing $O(n/B)$ pages. Our algorithms exploit the special features of non-redundant (minimal) reachability information, which we represent explicitly through the concept of expanded R-tuples. As in Chapter 4, the core of our data structure, is essentially a collection of non-redundant R-tuples, whose size (and that of the data structure itself) cannot exceed $O(n^2\tau)$. However, in our approach, all this space must be pre-allocated on disk. The benefit of our data structure is that algorithms explicitly manage data sequentially and, therefore, it is more suitable for

secondary memories in which random accesses are expensive.

Further investigations could be done toward improving the complexity of our update algorithm. Can `add_contact`(u, v, t) access less than $O(n^2\tau/B)$ pages? Another direction could be designing efficient disk-based data structures for the decremental and the fully-dynamic versions of this problem. With *unsorted* contact insertion and deletion, it seems to represent both a significant challenge and a natural extension of the present work, one that would certainly develop further our common understanding of temporal reachability. Finally, it could be worth to investigate compressing algorithms to reduce the space of our data structure and the number of pages accessed by our update algorithm. Specifically, we think that compression algorithms based on differences and run-length coding (DAMME et al., 2017) could achieve a very high compression rate since the arrays M_{out} and M_{in} store repeating ordered values. The compressing schema could also solve the pre-allocation and initialization problems since, initially, all cells of M_{out} and M_{in} have the same value, thus this configuration is very compressible.

Conclusion

The hypothesis of this thesis is: *using specialized data structures for both primary and secondary memory can improve the maintenance of reachability queries on large temporal graphs*. Even though the amount of researches on temporal graph is rapidly growing, only few of these have explored data structures to maintain and query reachability information. Therefore, throughout this document, we studied several dynamic data structures to incrementally maintain the reachability information as new contacts are added to large temporal graphs. These temporal graphs are large in the sense that they do not fit entirely primary memory, that is why we studied data structures for both primary and secondary memory. Data structures for secondary memory are used to persist the reachability information, whereas data structures for primary memory can be used to speed up the maintenance of part of this information. Each data structure proposed in this thesis contributed as a specific goal to answer our hypothesis. We highlight that all our data structures prioritize query over update time, thus they should be considered in scenarios where the number of queries is much higher than the number of contact insertions.

In Chapter 4, we studied a novel mathematical object called Timed Transitive Closure (TTC). A standard Transitive Closure (TC) describes whether a vertex can reach another through a path in a standard graph. Differently, a TTC describes whether a vertex can reach another through a journey in a temporal graph while tracking departure and arrival timestamps of a minimal set of journeys as time intervals. Then, we introduced a simple incremental data structure for primary memory based on our TTC that provides contact insertion in any arbitrary order and answers reachability queries. This data structure provides two types of query: the existential query, which checks whether a vertex can reach another through a journey within a time interval; and the constructive query, which, additionally, reconstructs an entire journey if such journey exists. This study concluded our first specific goal: *develop a base model and implement a simple data structure for primary memory capable of answering reachability queries on temporal graphs*.

In Chapter 5, we introduced new *compact* data structures to reduce the space usage in primary memory when maintaining reachability information. The main observation is that we can represent subsets of the time intervals present in a TTC as pairs of bit sequences, one for departure and the other for arrival timestamps. This arises from the fact that a TTC holds only information regarding non-nested time intervals for each of these subsets. Thus, our novel compact data structure uses dynamic bit-vectors to efficiently maintain pairs of bit sequences. We provided two data structure variants: one for temporally dense temporal graphs, which uses a raw representation of bits as base structure for dynamic bit-vectors; and the other for temporally sparse temporal graphs, which instead stores only distances between consecutive 1's with the cost of an additional encoding step. Additionally, we provided new algorithms for dynamic bit-vectors in order to speed up our compact data structure. Among these algorithms, we would like to highlight the `UNSET_ONE_RANGE(B, j_1, j_2)`, which clears the bits in the range $[\text{SELECT}_1(B, j_1), \text{SELECT}_1(B, j_2) - 1]$ of a dynamic bit-vector B using the split-join strategy. We showed through our experiments that our compact data structures reduce space considerably in several scenarios while having similar performance compared to our previous data structure. This study concluded our second specific goal: *implement a space efficient data structure for primary memory capable of efficiently answering reachability queries on temporal graphs.*

Finally, in Chapter 6, we studied a new data structure for secondary memory. Data structures for secondary memory must prioritize sequential reads and writes when possible due to the high latency of disk pages retrieval and the non-linear performance of random accesses. Following this advice, our novel data structure for secondary memory expands the information present in a TTC and organizes it as two arrays on second memory. Thanks to this data organization, all our algorithms accesses the TTC information sequentially. Moreover, as new contacts are inserted into our data structure, the average time to compute and aggregate new reachability information decreases since the search space of our algorithms reduces over time. As a result, even though our update algorithm is linear in τ , instead of logarithmic, we showed through our experiments that, in most cases, the average wall-clock time of inserting a sequence contacts is much lower than using a disk-based data structure following the data organization of our previous proposal. This study concluded our third specific goal: *implement a data structure for secondary memory capable of efficiently answering reachability queries on large temporal graphs.*

In addition to our main research, we also included in this document two appendices worth mentioning. Appendix A reviews static compact data structures for temporal graphs. Learning compact data structures was fundamental to complete this work, especially those for temporal graphs. Appendix B describes how to use the split-join strategy to remove a range of keys from B^+ -trees in time $O(\log_B \tau)$, where τ is the number of elements. The split-join strategy is commonly used to implement Binary Search Tree (BST)

algorithms in the context of parallel computation. We used this strategy to implement our baseline data structure described in Chapter 4, and we also used the general concept to improve one of our compact data structure in Chapter 5. To the best of our knowledge, no other research explicitly proposed the split and join algorithms for B⁺-trees.

Let us return to our hypothesis: *using specialized data structures for both primary and secondary memory can improve the maintenance of reachability queries on large temporal graphs*. In Chapter 4, we proposed a baseline dynamic data structure in primary memory to answer reachability queries in temporal graphs. However, this first data structure requires a prohibitive amount of space to store reachability information of large temporal graphs in primary memory. Then, in Chapter 5, we proposed a new compact data structure that considerably reduces the space usage in primary memory. This is very important to our scenario, where neither temporal graphs nor their reachability information fit entirely in primary memory. By reducing the space usage, we can maintain larger parts of the reachability information in primary memory and postpone/reduce accesses to secondary memory; therefore, processing the reachability information of large temporal graph becomes faster. Finally, in Chapter 6, we proposed a data structure for secondary memory that prioritizes sequential accesses in order to improve the performance of our algorithms. This data structure persists, as fast as possible, the reachability information that was previously computed in batches using our compact data structure for primary memory. Therefore, using the data structures we proposed in Chapters 5 and 6 effectively improves the maintenance of reachability queries on large temporal graphs.

7.1 List of publications

Journal publications:

- BRITO, L. F. A.; ALBERTINI, M. K.; CASTEIGTS, A.; TRAVENÇOLO, B. A. N. A dynamic data structure for temporal reachability with unsorted contact insertions. *Social Network Analysis and Mining*, v. 12, n. 1, p. 22, 2022.

Non peer-reviewed publications:

- BRITO, L. F. A.; ALBERTINI, M. K.; TRAVENÇOLO, B. A. N.; NAVARRO, G. A compact dynamic data structure for temporal reachability with unsorted contact insertions. URL: <<https://arxiv.org/abs/2308.11734>> (visited on 2023-08-25).
- BRITO, L. F. A.; ALBERTINI, M. K.; TRAVENÇOLO, B. A. N. A dynamic data structure for representing timed transitive closures on disk. 2022. URL: <<https://arxiv.org/abs/2306.13937>> (visited on 2023-08-22).

-
- BRITO, L. F. A.; ALBERTINI, M. K; TRAVENÇOLO, B. A. N. A Review of In-Memory Space-Efficient Data Structures for Temporal Graphs. 2022. URL: <<https://arxiv.org/abs/2204.12468>> (visited on 2023-06-03).

Bibliography

- ABEL, D. J. A B^+ -tree structure for large quadtrees. **Computer Vision, Graphics, and Image Processing**, Elsevier, v. 27, n. 1, p. 19–31, 1984. ISSN 0734-189X.
- ACKERMANN, W. Zum Hilbertschen aufbau der reellen zahlen. **Mathematische Annalen**, Springer-Verlag, v. 99, n. 1, p. 118–133, 1928.
- ADLER, M.; MITZENMACHER, M. Towards compressing web graphs. In: **Proceedings of the Data Compression Conference**. USA: IEEE Computer Society, 2001. (DCC '01), p. 203.
- AGARWAL, P. K.; ARGE, L.; PROCOPIUC, O.; VITTER, J. S. A framework for index bulk loading and dynamization. In: SPRINGER. **International Colloquium on Automata, Languages, and Programming**. Berlin, 2001. p. 115–127.
- AGARWAL, P. K.; ERICKSON, J. et al. Geometric range searching and its relatives. **Contemporary Mathematics**, Providence, RI: American Mathematical Society, v. 223, p. 1–56, 1999.
- AGRAWAL, R.; BORGIDA, A.; JAGADISH, H. V. Efficient management of transitive relationships in large data and knowledge bases. In: **Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data**. New York, NY, USA: Association for Computing Machinery, 1989. (SIGMOD 89), p. 253262.
- AJTAI, M.; FAGIN, R. Reachability is harder for directed than for undirected finite graphs. **The Journal of Symbolic Logic**, Association for Symbolic Logic, v. 55, n. 1, p. 113–150, 1990.
- ALBERT, R.; BARABÁSI, A.-L. Statistical mechanics of complex networks. **Rev. Mod. Phys.**, American Physical Society, v. 74, p. 47–97, 1 2002.
- AMER, A.; HOLLIDAY, J.; LONG, D. D. E.; MILLER, E. L.; PARIS, J.; SCHWARZ, T. Data management and layout for shingled magnetic recording. **IEEE Transactions on Magnetics**, v. 47, n. 10, p. 3691–3697, 2011.
- ANH, V. N.; MOFFAT, A. Inverted index compression using word-aligned binary codes. **Information Retrieval**, Springer, v. 8, n. 1, p. 151–166, 2005.

- ARGE, L. The buffer tree: A new technique for optimal I/O-algorithms. In: AKL, S. G.; DEHNE, F.; SACK, J.-R.; SANTORO, N. (Ed.). **Algorithms and Data Structures**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995. p. 334–345. ISBN 978-35404-4-7-4-7-4.
- ARGE, L.; DANNER, A.; TEH, S.-M. I/O-efficient point location using persistent B-trees. **ACM J. Exp. Algorithmics**, Association for Computing Machinery, New York, NY, USA, v. 8, p. 1.2es, dez. 2004. ISSN 1084-6654.
- ARGE, L.; VITTER, J. S. Optimal dynamic interval management in external memory. In: **Proceedings of 37th Conference on Foundations of Computer Science**. [S.l.: s.n.], 1996. p. 560–569.
- ATHANASSOULIS, M.; KESTER, M. S.; MAAS, L. M.; STOICA, R.; IDREOS, S.; AILAMAKI, A.; CALLAGHAN, M. Designing access methods: The RUM conjecture. In: **EDBT**. [S.l.: s.n.], 2016. v. 2016, p. 461–466.
- ATZORI, L.; IERA, A.; MORABITO, G.; NITTI, M. The social internet of things (SIoT) when social networks meet the internet of things: Concept, architecture and network characterization. **Computer Networks**, v. 56, n. 16, p. 3594–3608, 2012.
- BARJON, M.; CASTEIGTS, A.; CHAUMETTE, S.; JOHNNEN, C.; NEGGAZ, Y. M. **Testing Temporal Connectivity in Sparse Dynamic Graphs**. 2014. URL: <<https://arxiv.org/abs/1404.7634>> (visited on 2023-06-03).
- BAYER, R.; MCCREIGHT, E. Organization and maintenance of large ordered indices. In: **Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control**. New York, NY, USA: Association for Computing Machinery, 1970. (SIGFIDET '70), p. 107141. ISBN 978-145-037-9-4-1-0. Disponível em: <<https://doi.org/10.1145/1734663.1734671>>.
- BECKMANN, N.; KRIEGEL, H.-P.; SCHNEIDER, R.; SEEGER, B. The R*-tree: An efficient and robust access method for points and rectangles. **SIGMOD Rec.**, Association for Computing Machinery, New York, NY, USA, v. 19, n. 2, p. 322331, maio 1990.
- BEDOGNI, L.; FIORE, M.; GLACET, C. Temporal reachability in vehicular networks. In: **IEEE INFOCOM 2018 - IEEE Conference on Computer Communications**. [S.l.: s.n.], 2018. p. 81–89.
- BENTLEY, J. L. Decomposable searching problems. **Information Processing Letters**, v. 8, n. 5, p. 244–251, 1979. ISSN 0020-0190.
- BERNARDO, G. D. **New data structures and algorithms for the efficient management of large spatial datasets**. 2014. Doctoral dissertation at Coruña University. URL: <<http://hdl.handle.net/2183/13769>> (visited on 2023-06-03).
- BERNARDO, G. D.; BRISABOA, N. R.; CARO, D.; RODRÍGUEZ, M. A. Compact data structures for temporal graphs. In: IEEE. **Data Compression Conference (DCC), 2013**. [S.l.], 2013. p. 477–477.
- BESTA, M.; HOEFLER, T. **Survey and Taxonomy of Lossless Graph Compression and Space-Efficient Graph Representations**. 2019. URL: <<https://arxiv.org/abs/1806.01799>> (visited on 2023-06-03).

- BLELLOCH, G. E.; FERIZOVIC, D.; SUN, Y. Just join for parallel ordered sets. In: **Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures**. New York, NY, USA: Association for Computing Machinery, 2016. (SPAA '16), p. 253264. ISBN 978-145-034-2-1-0-0.
- BLONDEL, V. D.; GUILLAUME, J.-L.; LAMBIOTTE, R.; LEFEBVRE, E. Fast unfolding of communities in large networks. **Journal of Statistical Mechanics: Theory and Experiment**, IOP Publishing, v. 2008, n. 10, p. P10008, 10 2008.
- BRAMANDIA, R.; CHOI, B.; NG, W. K. On incremental maintenance of 2-hop labeling of graphs. In: **Proceedings of the 17th International Conference on World Wide Web**. New York, NY, USA: Association for Computing Machinery, 2008. (WWW '08), p. 845854.
- BRISABOA, N. R.; CARO, D.; FARIÑA, A.; RODRÍGUEZ, M. A. A compressed suffix-array strategy for temporal-graph indexing. In: SPRINGER. **International Symposium on String Processing and Information Retrieval**. [S.l.], 2014. p. 77–88.
- _____. Using compressed suffix-arrays for a compact representation of temporal-graphs. **Information Sciences**, Elsevier, v. 465, p. 459–483, 2018.
- BRISABOA, N. R.; IGLESIAS, E. L.; NAVARRO, G.; PARAMÁ, J. R. An efficient compression code for text databases. In: SPRINGER. **European Conference on Information Retrieval**. [S.l.], 2003. p. 468–481.
- BRITO, L. F. A.; ALBERTINI, M. K.; CASTEIGTS, A.; TRAVENÇOLO, B. A. N. A dynamic data structure for temporal reachability with unsorted contact insertions. **Social Network Analysis and Mining**, v. 12, n. 1, p. 22, 2022.
- BRITO, L. F. A.; ALBERTINI, M. K.; TRAVENÇOLO, B. A. N. **A dynamic data structure for representing timed transitive closures on disk**. 2023. URL: <<https://arxiv.org/abs/2306.13937>> (visited on 2023-08-22).
- BRITO, L. F. A.; ALBERTINI, M. K.; TRAVENÇOLO, B. A. N.; NAVARRO, G. **A compact dynamic data structure for temporal reachability with unsorted contact insertions**. 2023. URL: <<https://arxiv.org/abs/2308.11734>> (visited on 2023-08-25).
- BRITO, L. F. A.; TRAVENÇOLO, B. A. N.; ALBERTINI, M. K. **A review of in-memory space-efficient data structures for temporal graphs**. 2022. URL: <<https://arxiv.org/abs/2306.13937>> (visited on 2023-08-25).
- BRYCE, D.; KAMBHAMPATI, S. A tutorial on planning graph based reachability heuristics. **AI Magazine**, v. 28, n. 1, p. 47, mar. 2007.
- BUCHSBAUM, A. L.; GOLDWASSER, M. H.; VENKATASUBRAMANIAN, S.; WESTBROOK, J. On external memory graph traversal. In: **SODA '00**. [S.l.: s.n.], 2000.
- CACCIARI, L.; RAFIQ, O. A temporal reachability analysis. In: _____. **Protocol Specification, Testing and Verification XV: Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing**

and Verification, Warsaw, Poland, June 1995. Boston, MA: Springer US, 1996. p. 35–49. ISBN 978-038-734-8-9-2-6.

CAI, J.; POON, C. K. Path-hop: Efficiently indexing large graphs for reachability queries. In: **Proceedings of the 19th ACM International Conference on Information and Knowledge Management.** New York, NY, USA: Association for Computing Machinery, 2010. (CIKM '10), p. 119128.

CARO, D.; RODRÍGUEZ, M. A.; BRISABOIA, N. R. Data structures for temporal graphs based on compact sequence representations. **Information Systems**, Elsevier, v. 51, p. 1–26, 2015.

CARO, D.; RODRIGUEZ, M. A.; BRISABOIA, N. R.; FARINA, A. Compressed k^d -tree for temporal graphs. **Knowledge and Information Systems**, Springer, v. 49, n. 2, p. 553–595, 2016.

CASTEIGTS, A. **A Journey through Dynamic Networks (with Excursions).** 2018. Report for the authorization to direct research at Bordeaux University. URL: <<https://hal.science/tel-01883384>> (visited on 2023-06-03).

CASTEIGTS, A.; FLOCCHINI, P.; QUATTROCIOCCHI, W.; SANTORO, N. Time-varying graphs and dynamic networks. In: FREY, H.; LI, X.; RUEHRUP, S. (Ed.). **Ad-hoc, Mobile, and Wireless Networks.** Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. p. 346–359. ISBN 978-36422-2-4-5-0-8.

CHAMBI, S.; LEMIRE, D.; KASER, O.; GODIN, R. Better bitmap performance with roaring bitmaps. **Software: practice and experience**, Wiley Online Library, v. 46, n. 5, p. 709–719, 2016.

CHEN, L.; GUPTA, A.; KURUL, M. E. Stack-based algorithms for pattern matching on DAGs. In: **Proceedings of the 31st International Conference on Very Large Data Bases.** [S.l.]: VLDB Endowment, 2005. (VLDB '05), p. 493504. ISBN 1-59593-154-6.

CHEN, Y.; CHEN, Y. An efficient algorithm for answering graph reachability queries. In: **2008 IEEE 24th International Conference on Data Engineering.** [S.l.: s.n.], 2008. p. 893–902.

_____. Decomposing DAGs into spanning trees: A new way to compress transitive closures. In: **2011 IEEE 27th International Conference on Data Engineering.** [S.l.: s.n.], 2011. p. 1007–1018.

CHENG, J.; HUANG, S.; WU, H.; FU, A. W.-C. TF-Label: A topological-folding labeling scheme for reachability querying in a large graph. In: **Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data.** New York, NY, USA: Association for Computing Machinery, 2013. (SIGMOD '13), p. 193204.

CHENG, J.; YU, J. X.; LIN, X.; WANG, H.; YU, P. S. Fast computing reachability labelings for large graphs with high compression rate. In: **Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology.** New York, NY, USA: Association for Computing Machinery, 2008. (EDBT '08), p. 193204.

- CHENG, J.; YU, J. X.; TANG, N. Fast reachability query processing. In: LEE, M. L.; TAN, K.-L.; WUWONGSE, V. (Ed.). **Database Systems for Advanced Applications**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. p. 674–688. ISBN 978-35403-3-3-3-8-8.
- CIACCIA, P.; PATELLA, M.; ZEZULA, P. M-tree: An efficient access method for similarity search in metric spaces. In: **Proceedings of the 23rd International Conference on Very Large Data Bases**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. (VLDB '97), p. 426435. ISBN 1-55860-470-7.
- CLEMENTI, A. E. F.; MACCI, C.; MONTI, A.; PASQUALE, F.; SILVESTRI, R. Flooding time in edge-markovian dynamic graphs. In: **Proceedings of the Twenty-Seventh ACM Symposium on Principles of Distributed Computing**. New York, NY, USA: Association for Computing Machinery, 2008. (PODC '08), p. 213222.
- COHEN, E.; HALPERIN, E.; KAPLAN, H.; ZWICK, U. Reachability and distance queries via 2-hop labels. **SIAM Journal on Computing**, v. 32, n. 5, p. 1338–1355, 2003.
- COIMBRA, M. E.; FRANCISCO, A. P.; RUSSO, L. M. S.; BERNARDO, G. D.; LADRA, S.; NAVARRO, G. On dynamic succinct graph representations. In: **2020 Data Compression Conference (DCC)**. [S.l.: s.n.], 2020. p. 213–222.
- COMER, D. Ubiquitous B-tree. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 11, n. 2, p. 121137, jun. 1979. ISSN 0360-0300.
- DAMME, P.; HABICH, D.; HILDEBRANDT, J.; LEHNER, W. Lightweight data compression algorithms: An experimental survey (experiments and analyses). In: **EDBT**. [S.l.: s.n.], 2017.
- DATE, C. J. **An Introduction to Database Systems**. 8. ed. USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 0-321-19784-4.
- DING, Z.; GÜTING, R. H. Modeling temporally variable transportation networks. In: LEE, Y.; LI, J.; WHANG, K.-Y.; LEE, D. (Ed.). **Database Systems for Advanced Applications**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. p. 154–168.
- ELIAS, P. Universal codeword sets and representations of the integers. **IEEE Transactions on Information Theory**, v. 21, n. 2, p. 194–203, 1975. ISSN 1557-9654.
- ENRIGHT, J.; MEEKS, K.; MERTZIOS, G. B.; ZAMARAEV, V. Deleting edges to restrict the size of an epidemic in temporal networks. **Journal of Computer and System Sciences**, v. 119, p. 60–77, 2021. ISSN 0022-0000.
- FAGIN, R.; NIEVERGELT, J.; PIPPENGER, N.; STRONG, H. R. Extendible hashing: a fast access method for dynamic files. **ACM Trans. Database Syst.**, Association for Computing Machinery, New York, NY, USA, v. 4, n. 3, p. 315344, set. 1979.
- GAGIE, T.; NAVARRO, G.; PUGLISI, S. J. New algorithms on wavelet trees and applications to information retrieval. **Theoretical Computer Science**, Elsevier, v. 426, p. 25–41, 2012.

GEORGE, B.; KIM, S.; SHEKHAR, S. Spatio-temporal network databases and routing algorithms: A summary of results. In: PAPADIAS, D.; ZHANG, D.; KOLLIOS, G. (Ed.). **Advances in Spatial and Temporal Databases**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 460–477. ISBN 978-35407-3-5-4-0-3.

GRINSTEAD, C. M.; SNELL, J. L. **Introduction to probability**. [S.l.]: American Mathematical Soc., 1997.

GROSSI, R.; GUPTA, A.; VITTER, J. S. High-order entropy-compressed text indexes. In: SOCIETY FOR INDUSTRIAL AND APPLIED MATHEMATICS. **Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms**. [S.l.], 2003. p. 841–850.

GROSSI, R.; VITTER, J. S. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. **SIAM Journal on Computing**, SIAM, v. 35, n. 2, p. 378–407, 2005.

HAN, W.; MIAO, Y.; LI, K.; WU, M.; YANG, F.; ZHOU, L.; PRABHAKARAN, V.; CHEN, W.; CHEN, E. Chronos: A graph engine for temporal graph analysis. In: **ACM. Proceedings of the Ninth European Conference on Computer Systems**. [S.l.], 2014. p. 1.

HASAN, K. T.; NOORI, S. R. H.; SALAM, A.; KABIR, M. A. Making sense of time: timeline visualization for public transport schedule. In: **Symposium on Human-Computer Interaction and Information Retrieval (HCIR 2011), Washington**. [S.l.: s.n.], 2011.

HIRVISALO, V.; NUUTILA, E.; SOISALON-SOININEN, E. **Transitive closure algorithm DISK TC and its performance analysis**. 1996. Technical report at Helsinki University of Technology. URL: <<http://www.cs.hut.fi/~enu/tc.html>> (visited on: 2020-04-02).

HOLM, J.; LICHTENBERG, K. D.; THORUP, M. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. **Journal of the ACM (JACM)**, ACM New York, NY, USA, v. 48, n. 4, p. 723–760, 2001.

HONO, K.; TAKAHASHI, Y.; JU, G.; THIELE, J.-U.; AJAN, A.; YANG, X.; RUIZ, R.; WAN, L. Heat-assisted magnetic recording media materials. **MRS Bulletin**, Springer, v. 43, n. 2, p. 93–99, 2018.

HURTER, C.; ERSOY, O.; FABRIKANT, S. I.; KLEIN, T. R.; TELEA, A. C. Bundled visualization of dynamic graph and trail data. **IEEE Transactions on Visualization and Computer Graphics**, v. 20, n. 8, p. 1141–1157, 2014.

ITALIANO, G. F. Amortized efficiency of a path retrieval data structure. **Theoretical Computer Science**, Elsevier, v. 48, p. 273–281, 1986.

_____. Finding paths and deleting edges in directed acyclic graphs. **Information Processing Letters**, v. 28, n. 1, p. 5–11, 1988. ISSN 0020-0190.

JACOBSON, G. J. **Succinct static data structures**. Pittsburgh, PA, USA: [s.n.], 1988. Doctoral dissertation at Carnegie Mellon University.

- JAGADISH, H. V. A compression technique to materialize transitive closure. **ACM Trans. Database Syst.**, Association for Computing Machinery, New York, NY, USA, v. 15, n. 4, p. 558598, dez. 1990.
- JIN, R.; WANG, G. Simple, fast, and scalable reachability oracle. **Proc. VLDB Endow.**, VLDB Endowment, v. 6, n. 14, p. 19781989, 2013. ISSN 2150-8097.
- JOHNSON, T.; SHASHA, D. 2Q: A low overhead high performance buffer management replacement algorithm. In: **Proceedings of the 20th International Conference on Very Large Data Bases**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994. (VLDB '94), p. 439–450.
- KHURANA, U.; DESHPANDE, A. Efficient snapshot retrieval over historical graph data. In: IEEE. **2013 IEEE 29th International Conference on Data Engineering (ICDE)** . [S.l.], 2013. p. 997–1008.
- KING, V.; SAGERT, G. A fully dynamic algorithm for maintaining the transitive closure. **Journal of Computer and System Sciences**, v. 65, n. 1, p. 150–167, 2002.
- KNUTH, D. E. Dynamic huffman coding. **Journal of Algorithms**, v. 6, n. 2, p. 163–180, 1985. ISSN 0196-6774.
- KRYDER, M. H.; GAGE, E. C.; MCDANIEL, T. W.; CHALLENGER, W. A.; ROTTMAYER, R. E.; JU, G.; HSIA, Y.; ERDEN, M. F. Heat assisted magnetic recording. **Proceedings of the IEEE**, v. 96, n. 11, p. 1810–1835, 2008.
- LABOUSEUR, A. G.; BIRNBAUM, J.; OLSEN, P. W.; SPILLANE, S. R.; VIJAYAN, J.; HWANG, J.-H.; Han , W.-S. The g^* graph database: Efficiently managing large distributed dynamic graphs. **Distributed and Parallel Databases**, Springer, v. 33, n. 4, p. 479–514, 2015.
- ŁĄCKI, J. Improved deterministic algorithms for decremental reachability and strongly connected components. **ACM Transactions on Algorithms (TALG)**, ACM New York, NY, USA, v. 9, n. 3, p. 1–15, 2013.
- LATAPY, M.; VIARD, T.; MAGNIEN, C. Stream graphs and link streams for the modeling of interactions over time. **Social Network Analysis and Mining**, Springer, v. 8, n. 1, p. 1–29, 2018.
- LEMIRE, D.; BOYTSOV, L. Decoding billions of integers per second through vectorization. **Software: Practice and Experience**, John Wiley & Sons, Inc., USA, v. 45, n. 1, p. 129, 2015. ISSN 0038-0644.
- LIBEN-NOWELL, D.; KLEINBERG, J. The link-prediction problem for social networks. **Journal of the American society for information science and technology**, Wiley Online Library, v. 58, n. 7, p. 1019–1031, 2007.
- LINHARES, C. D. G.; PONCIANO, J. R.; PAIVA, J. G. S.; TRAVENÇOLO, B. A. N.; ROCHA, L. E. C. Visualisation of structure and processes on temporal networks. In: HOLME, P.; SARAMÄKI, J. (Ed.). **Computational Social Sciences**. Cham: Springer International Publishing, 2019. p. 83–105. ISBN 978-30302-3-4-9-5-9.

- LINHARES, C. D. G.; TRAVEÇOLO, B. A. N.; PAIVA, J. G. S.; ROCHA, L. E. C. DyNetVis: a system for visualization of dynamic networks. In: **ACM. Proceedings of the Symposium on Applied Computing**. [S.l.], 2017. p. 187–194.
- LITWIN, W. Linear hashing: A new tool for file and table addressing. In: **Proceedings of the Sixth International Conference on Very Large Data Bases - Volume 6**. [S.l.]: VLDB Endowment, 1980. (VLDB '80), p. 212223.
- LYU, Q.; LI, Y.; HE, B.; GONG, B. DBL: Efficient reachability queries on dynamic graphs. In: JENSEN, C. S.; LIM, E.-P.; YANG, D.-N.; LEE, W.-C.; TSENG, V. S.; KALOGERAKI, V.; HUANG, J.-W.; SHEN, C.-Y. (Ed.). **Database Systems for Advanced Applications**. Cham: Springer International Publishing, 2021. p. 761–777. ISBN 978-30307-3-1-9-7-7.
- MALEWICZ, G.; AUSTERN, M. H.; BIK, A. J. C.; DEHNERT, J. C.; HORN, I.; LEISER, N.; CZAJKOWSKI, G. Pregel: A system for large-scale graph processing. In: **Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data**. New York, NY, USA: ACM, 2010. (SIGMOD '10), p. 135–146.
- MANBER, U.; MYERS, G. Suffix arrays: a new method for on-line string searches. **siam Journal on Computing**, SIAM, v. 22, n. 5, p. 935–948, 1993.
- MARTENSEN, A. C.; SAURA, S.; FORTIN, M.-J. Spatio-temporal connectivity: assessing the amount of reachable habitat in dynamic landscapes. **Methods in Ecology and Evolution**, v. 8, n. 10, p. 1253–1264, 2017.
- MATHIEU, C.; RAJARAMAN, R.; YOUNG, N. E.; YOUSEFI, A. Competitive data-structure dynamization. In: SIAM. **Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)**. [S.l.], 2021. p. 2269–2287.
- MICHAIL, O. An introduction to temporal graphs: An algorithmic perspective. **Internet Mathematics**, Taylor & Francis, v. 12, n. 4, p. 239–280, 2016.
- MOFFITT, V. Z.; STOYANOVICH, J. **Querying Evolving Graphs with Portal**. 2016. URL: <<https://arxiv.org/abs/1602.00773>> (visited on 2023-06-03).
- NAVARRO, G. **Compact Data Structures: A Practical Approach**. 1st. ed. USA: Cambridge University Press, 2016. ISBN 1-107-15238-0.
- NEUMANN, T.; WEIKUM, G. x-RDF-3X: fast querying, high update rates, and consistency for RDF databases. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 3, n. 1-2, p. 256–263, 2010.
- NUUTILA, E. Efficient transitive closure computation in large digraphs. **Acta Polytechnica Scandinavia: Math. Comput. Eng.**, The Finnish Academy of Technology, FIN, v. 74, p. 1124, jul. 1995.
- OVERMARS, M. H. The design of dynamic data structures. In: **Lecture Notes in Computer Science**. [S.l.]: Springer Berlin, Heidelberg, 1987. ISSN 0302-9743.
- PONCIANO, J. R.; VEZONO, G. P.; LINHARES, C. D. G. Simulating and visualizing infection spread dynamics with temporal networks. In: **Anais do XXXVI Simpósio Brasileiro de Banco de Dados (SBBDD 2021)**. [S.l.]: Sociedade Brasileira de Computação - SBC, 2021.

- PREZZA, N. A framework of dynamic data structures for string processing. In: ILIOPOULOS, C. S.; PISSIS, S. P.; PUGLISI, S. J.; RAMAN, R. (Ed.). **16th International Symposium on Experimental Algorithms (SEA 2017)**. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017. (Leibniz International Proceedings in Informatics (LIPIcs), v. 75), p. 11:1–11:15. ISBN 978-39597-7-0-3-6-1. ISSN 1868-8969.
- QIAO, M. Query processing in large-scale networks. In: . [S.l.: s.n.], 2013.
- QIAO, M.; CHENG, H.; QIN, L.; YU, J. X.; YU, P. S.; CHANG, L. Computing weight constraint reachability in large networks. **The VLDB Journal**, v. 22, p. 275–294, 2012.
- RAMAN, R.; RAMAN, V.; SATTI, S. R. Succinct indexable dictionaries with applications to encoding k -ary trees, prefix sums and multisets. **ACM Transactions on Algorithms (TALG)**, ACM New York, NY, USA, v. 3, n. 4, p. 43–es, 2007.
- RODITTY, L. A faster and simpler fully dynamic transitive closure. **ACM Transactions on Algorithms (TALG)**, ACM New York, NY, USA, v. 4, n. 1, p. 1–16, 2008.
- RODITTY, L.; ZWICK, U. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. **SIAM Journal on Computing**, v. 45, n. 3, p. 712–733, 2016.
- ROY, A.; MIHAILOVIC, I.; ZWAENEPOEL, W. X-Stream: Edge-centric graph processing using streaming partitions. In: **Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles**. New York, NY, USA: ACM, 2013. (SOSP '13), p. 472–488.
- ROZENSHTAIN, P.; GIONIS, A.; PRAKASH, B. A.; VREEKEN, J. Reconstructing an epidemic over time. In: **Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining**. New York, NY, USA: Association for Computing Machinery, 2016. (KDD '16), p. 18351844. ISBN 978-145-034-2-3-2-2.
- SADAKANE, K. New text indexing functionalities of the compressed suffix arrays. **Journal of Algorithms**, v. 48, n. 2, p. 294–313, 2003.
- SAHU, S.; MHEDHBI, A.; SALIHOGLU, S.; LIN, J.; ÖZSU, M. T. The ubiquity of large graphs and surprising challenges of graph processing. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 11, n. 4, p. 420–431, dez. 2017.
- SCHAIK, S. J. van; MOOR, O. de. A memory efficient reachability data structure through bit vector compression. In: **Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data**. New York, NY, USA: Association for Computing Machinery, 2011. (SIGMOD '11), p. 913924.
- SEAGATE. **HAMR Technology**. <https://www.seagate.com/files/www-content/innovation/hamr/_shared/files/tp707-1-1712us-hamr.pdf>.
- _____. **Mach.2 Technology**. <<https://www.seagate.com/files/www-content/solutions/mach-2-multi-actuator-hard-drive/files/tp714-dot-2-2006us-mach-2-technology-paper.pdf>>.

- SEUFERT, S.; ANAND, A.; BEDATHUR, S.; WEIKUM, G. FERRARI: Flexible and efficient reachability range assignment for graph indexing. In: **2013 IEEE 29th International Conference on Data Engineering (ICDE)** . [S.l.: s.n.], 2013. p. 1009–1020.
- SHILOACH, Y.; EVEN, S. An on-line edge-deletion problem. **Journal of the ACM (JACM)**, ACM New York, NY, USA, v. 28, n. 1, p. 1–4, 1981.
- SHIRANI-MEHR, H.; KASHANI, F. B.; SHAHABI, C. Efficient reachability query evaluation in large spatiotemporal contact datasets. **Proc. VLDB Endow.**, v. 5, p. 848–859, 2012.
- STRZHELETSKA, E. V. **Efficient Processing of Novel Reachability-Based Queries on Large Spatiotemporal Datasets**. 2018. Doctoral dissertation at University of California Riverside. URL: <<https://escholarship.org/uc/item/2j83p0bg>> (visited on 2023-06-03).
- STRZHELETSKA, E. V.; TSOTRAS, V. Efficient processing of reachability queries with meetings. **Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems**, 2017.
- TANG, J.; MUSOLESI, M.; MASCOLO, C.; LATORA, V. Characterising temporal distance and reachability in mobile and online social networks. **SIGCOMM Computer Communication Review**, Association for Computing Machinery, New York, NY, USA, v. 40, n. 1, p. 118124, 2010. ISSN 0146-4833.
- TARJAN, R. Depth-first search and linear graph algorithms. **SIAM Journal on Computing**, v. 1, n. 2, p. 146–160, 1972.
- TARJAN, R. E. Efficiency of a good but not linear set union algorithm. **J. ACM**, Association for Computing Machinery, New York, NY, USA, v. 22, n. 2, p. 215225, abr. 1975.
- _____. A class of algorithms which require nonlinear time to maintain disjoint sets. **Journal of computer and system sciences**, Elsevier, v. 18, n. 2, p. 110–127, 1979.
- VELOSO, R. R.; CERF, L.; JR, W. M.; ZAKI, M. J. Reachability queries in very large graphs: A fast refined online search approach. In: **International Conference on Extending Database Technology**. [S.l.: s.n.], 2014. p. 511–522.
- WANG, H.; HE, H.; YANG, J.; YU, P. S.; YU, J. X. Dual labeling: Answering graph reachability queries in constant time. In: **22nd International Conference on Data Engineering (ICDE'06)**. [S.l.: s.n.], 2006. p. 75–75.
- WANG, S.; LIN, W.; YANG, Y.; XIAO, X.; ZHOU, S. Efficient route planning on public transportation networks: A labelling approach. In: **Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data**. New York, NY, USA: Association for Computing Machinery, 2015. (SIGMOD '15), p. 967982.
- WEI, H.; YU, J. X.; LU, C.; JIN, R. Reachability querying: an independent permutation labeling approach. **The VLDB Journal**, Springer, v. 27, n. 1, p. 1–26, 2018.

- WHITBECK, J.; AMORIM, M. D. de; CONAN, V.; GUILLAUME, J.-L. Temporal reachability graphs. In: **Proceedings of the 18th Annual International Conference on Mobile Computing and Networking**. New York, NY, USA: Association for Computing Machinery, 2012. (Mobicom '12), p. 377388. ISBN 978-145-031-1-5-9-5.
- WILLIAMS, M. J.; MUSOLESI, M. Spatio-temporal networks: reachability, centrality and robustness. **Royal Society Open Science**, v. 3, n. 6, p. 160196, 2016.
- WU, G.; DING, Y.; LI, Y.; BAO, J.; ZHENG, Y.; LUO, J. Mining spatio-temporal reachable regions over massive trajectory data. In: **2017 IEEE 33rd International Conference on Data Engineering (ICDE)**. [S.l.: s.n.], 2017. p. 1283–1294. ISSN 2375-026X.
- WU, H.; HUANG, Y.; CHENG, J.; LI, J.; KE, Y. Reachability and time-based path queries in temporal graphs. In: **2016 IEEE 32nd International Conference on Data Engineering (ICDE)**. [S.l.: s.n.], 2016. p. 145–156.
- XIAO, H.; ASLAY, C.; GIONIS, A. Robust cascade reconstruction by steiner tree sampling. In: **2018 IEEE International Conference on Data Mining (ICDM)**. [S.l.: s.n.], 2018. p. 637–646.
- XIAO, H.; ROZENSHTEIN, P.; TATTI, N.; GIONIS, A. Reconstructing a cascade from temporal observations. In: _____. **Proceedings of the 2018 SIAM International Conference on Data Mining (SDM)**. [S.l.: s.n.], 2018. p. 666–674.
- XUAN, B. B.; FERREIRA, A.; JARRY, A. Computing shortest, fastest, and foremost journeys in dynamic networks. **International Journal of Foundations of Computer Science**, World Scientific, v. 14, n. 02, p. 267–285, 2003.
- _____. Computing shortest, fastest, and foremost journeys in dynamic networks. **International Journal of Foundations of Computer Science**, v. 14, n. 02, p. 267–285, 2003.
- YANG, T.; CHI, Y.; ZHU, S.; GONG, Y.; JIN, R. Detecting communities and their evolutions in dynamic social networks a bayesian approach. **Machine learning**, Springer, v. 82, n. 2, p. 157–189, 2011.
- YANO, Y.; AKIBA, T.; IWATA, Y.; YOSHIDA, Y. Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In: **Proceedings of the 22nd ACM International Conference on Information and Knowledge Management**. New York, NY, USA: Association for Computing Machinery, 2013. (CIKM '13), p. 16011606.
- YILDIRIM, H.; CHAOJI, V.; ZAKI, M. J. GRAIL: a scalable index for reachability queries in very large graphs. **The VLDB Journal**, Springer, v. 21, n. 4, p. 509–534, 2012.
- YU, J. X.; CHENG, J. Graph reachability queries: a survey. In: _____. **Managing and Mining Graph Data**. Boston, MA: Springer US, 2010. p. 181–215.
- ZENG, W.; FU, C.-W.; ARISONA, S. M.; ERATH, A.; QU, H. Visualizing mobility of public transportation system. **IEEE Transactions on Visualization and Computer Graphics**, v. 20, n. 12, p. 1833–1842, 2014. ISSN 1941-0506.

ZHANG, Y.; LIAO, X.; SHI, X.; JIN, H.; HE, B. Efficient disk-based directed graph processing: A strongly connected component approach. **IEEE Transactions on Parallel and Distributed Systems**, v. 29, n. 4, p. 830–842, 2018.

ZHANG, Z.; YU, J. X.; QIN, L.; ZHU, Q.; ZHOU, X. I/O cost minimization: reachability queries processing over massive graphs. In: **EDBT '12**. [S.l.: s.n.], 2012.

_____. I/O cost minimization: Reachability queries processing over massive graphs. In: **Proceedings of the 15th International Conference on Extending Database Technology**. New York, NY, USA: Association for Computing Machinery, 2012. (EDBT '12), p. 468479.

ZHU, A. D.; LIN, W.; WANG, S.; XIAO, X. Reachability queries on large dynamic graphs: A total order approach. In: **Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data**. New York, NY, USA: Association for Computing Machinery, 2014. (SIGMOD '14), p. 13231334.

ZHU, J.; ZHU, X.; TANG, Y. Microwave assisted magnetic recording. **IEEE Transactions on Magnetics**, v. 44, n. 1, p. 125–131, 2008.

ZUKOWSKI, M.; HEMAN, S.; NES, N.; BONCZ, P. Super-scalar RAM-CPU cache compression. In: IEEE. **Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on**. [S.l.], 2006. p. 59–59.

Appendix

Space-Efficient Data Structures for Querying Temporal Graphs in Primary Memory

In this chapter, we present a review about compact data structures to store and query large temporal graphs in primary memory, which was originally written in one of our papers entitled “A Review of In-Memory Space-Efficient Data Structures for Temporal Graphs” submitted in June 4th, 2020 to the Elsevier journal “Information Systems” and currently being peer-reviewed. These specialized data structures provide useful queries while spending little space as possible. Some of them store a compressed version of data. However, they compute queries without decompressing the data (BRISABOA et al., 2014; CARO; RODRÍGUEZ; BRISABOA, 2015; CARO et al., 2016; BRISABOA et al., 2018). The literature calls these approaches self-indexed space-efficient data structures.

For example, Grossi, Gupta e Vitter (2003) introduced the wavelet tree data structure that stores a list of contacts as a sequence of n symbols belonging to an alphabet of size $\sigma = |V| + |T|$ using only $n \lceil \log \sigma \rceil$ bits. The wavelet tree executes fundamental queries such as determining the frequency of symbols in a sub-range of the sequence in $O(\log \sigma)$ time. By using the wavelet tree, some data structures can quickly answer low-level queries for temporal graphs.

In the next sections, we detail the following structures to index temporal graphs using compression techniques: time-interval Log per Edge (EdgeLog) (XUAN; FERREIRA; JARRY, 2003a), adjacency Log of Events (EveLog) (CARO; RODRÍGUEZ; BRISABOA, 2015), Compact Adjacency Sequence (CAS) (CARO; RODRÍGUEZ; BRISABOA, 2015), Compressed Events ordered by Time (CET) (CARO; RODRÍGUEZ; BRISABOA, 2015), Temporal Graph Compressed Suffix Array (TGCSA) (BRISABOA et al., 2018), and Compressed k^d -tree (CARO et al., 2016). For clarity, in the following sections we explain how some fundamental queries work on these data structures after they are built.

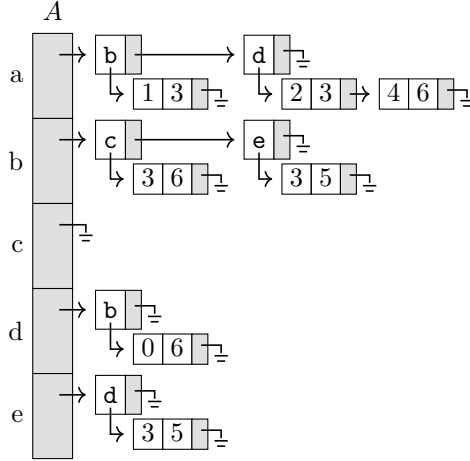


Figure 19 – EdgeLog representation of the temporal graph shown in Figure 2. EdgeLog structure stores an array A containing adjacency lists indexed by source vertex u . Each adjacency list $A[u]$ has target vertices v . For each vertex v in $A[u]$, there is a list of intervals during which the edge (u, v) is active. EdgeLog compresses these lists with DeltaGap and stores them in a contiguous space, instead of a list structure with pointers.

The content present in this chapter was published on the arXiv repository (BRITO; TRAVENÇOLO; ALBERTINI, 2022) available at <https://arxiv.org/abs/2204.12468>.

A.1 Time-Interval Log per Edge

Time-interval Log per Edge (EdgeLog) keeps a list of time intervals for each edge (XUAN; FERREIRA; JARRY, 2003a). As shown in Figure 19, this structure organizes contacts similarly to an inverted index (ANH; MOFFAT, 2005) with three levels. In the base level, it has an array A indexed by source vertices u containing pointers to adjacency lists. In the first level, each adjacency list stores target vertices v and another list containing, in the second level, the time intervals in which edges (u, v) are active. This technique compresses both adjacency and interval lists in order to reduce space. First, each list is kept sorted to speed up queries by using binary search and to represent sequences of vertex and timestamps identifiers as DeltaGap encodings (ADLER; MITZENMACHER, 2001). This encoding transforms sequences of numbers $L = l_1, l_2, \dots, l_n$ on differences of subsequent numbers in the form $D = l_2 - l_1, l_3 - l_2, \dots, l_n - l_{n-1}$. Then, variable bit-wise compression techniques, such as Huffman Code (KNUTH, 1985), or word-wise compression techniques, such as PForDelta (ZUKOWSKI et al., 2006) are applied to compress DeltaGap encodings.

This strategy reduces space usage because D has lower entropy than L and represents resulting values with fewer bits due to the lower range of the numbers. For commonly used encoding techniques for graphs we suggest the survey by Besta e Hoeffler (2019) and

the textbook by Navarro (2016).

A.1.1 Operation `has_edge`

In EdgeLog, an algorithm to answer `has_edge`(u, v, t_1, t_2) first decompresses the adjacency list $A[u]$ into a contiguous space in memory, then it performs a binary search to find v and the corresponding compressed list of intervals T_v . Next, it decompresses T_v and checks if the interval $[t_{begin}, t_{end}]$ overlaps with some interval in T_v by performing a second binary search. If so, an edge (u, v) exists during $[t_{begin}, t_{end}]$.

A.1.2 Operation `neighbors`

Similarly to the `has_edge` operation, an algorithm to answer `neighbors`(u, t_1, t_2) iterates all target vertices v' in $A[u]$ and checks if the list of intervals $T_{v'}$ correspond to interval $[t_{begin}, t_{end}]$, according to weak or strong semantics. If the query has weak semantic, then it checks if $[t_{begin}, t_{end}]$ overlaps with some $[t'_{begin}, t'_{end}] \in T_{v'}$. Otherwise, if it has strong semantic, then the algorithm checks if there is some interval in $T_{v'}$ such that $t'_{begin} \leq t_{begin} < t_{end} \leq t'_{end}$. The result is a list containing all neighbors that satisfy these conditions.

A.1.3 Operation `neighborsr`

EdgeLog indexes edges only by source vertices, therefore, there is no efficient way to compute reverse queries such as `neighborsr`(u, t_1, t_2).

A naïve algorithm decompresses every list in first level and test the interval conditions on every $(v, T_v) \in A[u]$ for all $u' \in V$. Another approach is to keep a second EdgeLog structure that indexes edges by target vertices instead. Therefore, an algorithm to answer `neighborsr`(u, t_1, t_2) runs `neighbors`(u, t_1, t_2) on this second structure. However, this approach doubles the space required to store temporal graphs.

A.2 Adjacency Log of Events

Adjacency Log of Events (EveLog) data structure stores events about edge activation and deactivation (CARO; RODRÍGUEZ; BRISABOA, 2015). As shown in Figure 20, EveLog has an array E indexed by source vertices u containing pointers to lists of events ordered by time. Each list has events $(v, t) \in V \times T$ that represent activation or deactivation of edge (u, v) at time t . EveLog does not store explicitly whether events represent activation or deactivation. Instead, it uses a parity property (CARO; RODRÍGUEZ; BRISABOA, 2015). As $E[u]$ is time-ordered, we need only to count the number of occur-

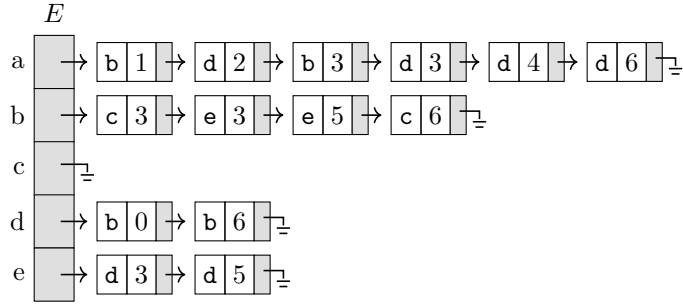


Figure 20 – EveLog representation of the temporal graph shown in Figure 2. This structure has an array E containing event lists indexed by source vertices u . Each event list $E[u]$ has pairs (v, t) that represent events of activation or deactivation of edge (u, v) at time t . At implementation level, EveLog separates $E[u]$ in two lists, V and T , containing target vertices and timestamps. It compresses the ordered T lists using DeltaGap and V lists using ETDC. Then it stores them in a contiguous space instead of a list structure with pointers. Note that we can check if (u, v) is active at timestamp t by determining the frequency of v symbols in $E[u]$ until the last t symbol.

rences of target vertices v until some timestamp t , if it is odd, then edge (u, v) is active at t , otherwise, it is not.

Each list of events is compressed to reduce space. First, EveLog separates lists $E[u]$ containing elements (v, t) in two different lists $\mathcal{V} = v_1, v_2, \dots, v_k$ and $\mathcal{T} = t_1, t_2, \dots, t_k$. As \mathcal{T} is ordered, EveLog applies DeltaGap compression, the same approach used to compress EdgeLog lists. However, \mathcal{V} is not ordered and EveLog cannot sort separately, otherwise, EveLog would lose the mapping between pairs on \mathcal{V} and \mathcal{T} . Therefore, DeltaGap encodings cannot be used to decrease the entropy of \mathcal{V} lists. Caro, Rodríguez e Brisaboa (2015) chose the End-Tagged Dense Codes (ETDC) (BRISABOA et al., 2003) technique, often used in information retrieval context, to compress \mathcal{V} lists since it is faster than Huffman Code while producing compressed sequences only 2.5% bigger.

A.2.1 Operation `has_edge`

An algorithm to answer $has_edge(\mathcal{G}, u, v, t_{begin}, t_{end})$ first decompresses both lists \mathcal{V} and \mathcal{T} associated with $E[u]$. Then, it performs binary searches in \mathcal{T} to find the positions i and j associated with the last timestamp symbols of t_{begin} and t_{end} , respectively. Next, it finds the frequencies f_{begin} and f_{end} of elements v inside the subsequences $\mathcal{V}[1, i] = v_1, v_2, \dots, v_i$ and $\mathcal{V}[i, j] = v_i, v_{i+1}, \dots, v_j$. If f_{begin} is odd, then edge (u, v) is active at time t_{begin} , and if f_{end} is greater than zero, then (u, v) is active at some timestamp during the interval $[t_{begin}, t_{end}]$, otherwise is not active. Note that, if it is a point-based query, $t = t_{begin} = t_{end}$ and $f = f_{begin} = f_{end}$, then the algorithm only needs to check if f is odd or even to answer whether (u, v) is, respectively, active or not at timestamp t .

A.2.2 Operation neighbors

An algorithm to answer $neighbors(\mathcal{G}, u, t_{begin}, t_{end})$ performs a similar approach, however, it counts frequencies for each possible vertex v' . First, it finds the frequencies f_{begin} and f_{end} for all vertices v' in both subsequences $\mathcal{V}[1, i]$ and $\mathcal{V}[i, j]$, where i and j are positions in \mathcal{T} associated with times t_{begin} and t_{end} , respectively. Next, depending on the query, it checks weak or strong semantics. If $neighbors(\mathcal{G}, u, t_{begin}, t_{end})$ has weak semantic, then the algorithm retrieves all contacts in which edge (u, v') is active at any timestamp during $[t_{begin}, t_{end}]$. Therefore, it returns a contact if edge (u, v') enters active in $[t_{begin}, t_{end}]$ — same as f_{begin} being odd — or if it enters inactive and, later, it activates at some timestamp during $[t_{begin}, t_{end}]$ — same as f_{end} being greater than 0. Instead, if the query has strong semantic, then it only retrieves contacts in which edge (u, v') is active during the interval $[t_{begin}, t_{end}]$. Therefore, it returns a contact if edge (u, v') enters active — same as f_{begin} being odd — and keeps active until t_{end} — same as f_{end} being equals to 0.

A.2.3 Operation neighbors^r

EveLog cannot answer $neighbors^r(\mathcal{G}, v, t_{begin}, t_{end})$ efficiently. An algorithm would decompress every event list and, for each one, it would count frequencies of edge events linearly until time t . As in EdgeLog, a workaround is to build a second EveLog structure and use it to run the direct query instead. However, as EveLog stores every event, the negative impact of doubling the space is higher than for the EdgeLog structure.

A.3 Compact Adjacency Sequence

Compact Adjacency Sequence (CAS) also stores activation and deactivation events and uses the parity property to answer queries. It determines the frequency of edge events in logarithmic time by using a wavelet tree (CARO; RODRÍGUEZ; BRISABOA, 2015).

As illustrated in Figure 21, CAS represents a temporal graph as a sequence $S = s_1, s_2, \dots, s_n$ of symbols $s_i \in T \cup V$ and an extra bitmap B of size $n + |V|$ for storage of information about adjacencies.

CAS partitions the sequence S into blocks representing events $(t, v) \in T \times V$ associated with each source vertex $u \in V$. The partitioning of S follows two steps: (1) sort the events (t, v) by the source vertices and by their timestamps, respectively; and (2) remove the timestamp repetitions.

The bitmap B marks the starting position k_i of the block associated with each i -th source vertex in S by filling with k_i 0's after every i -th bit set to 1. In other words, the value of the first position of B is 1 and the values of the k_1 subsequent positions,

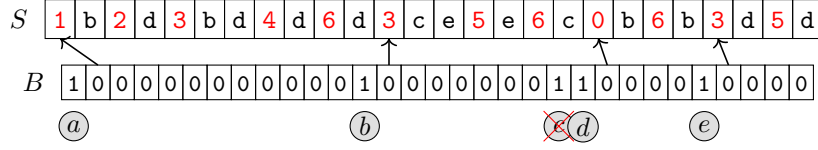


Figure 21 – CAS representation of the temporal graph shown in Figure 2. CAS structure stores a sequence S and a bitmap B . The u -th bit set to 1 in B marks the beginning of the S block associated with the u -th source vertex. Each block associated with the u -th source vertex is an event list ordered by time where, after a timestamp symbol t , there are target vertices symbols v that, together, represent events of activation or deactivation of edges (u, v) at timestamp t . The arrows illustrate the beginning of each u -th block in S and the first corresponding 0 in B . The vertex c is red-crossed because no event leaves it.

associated with the first source vertex, are 0. The value of the next initialized position is 1 and the values of the following k_2 positions, associated with the second source vertex, are 0. This pattern continues for each source vertex. Note that CAS uses a sorted set V . B is implemented as a special abstract data type named bitvector, which provides two operations: $c = \text{rank}(B, s, i)$ and $j = \text{select}(B, s, c)$, enabling us to compute the count of a given symbol s up to a position i and to get the smallest position j with count c for symbol s (NAVARRO, 2016). RRR is a well-known example¹ of succinct data structure to implement a bitvector with $\text{rank}()$ and $\text{select}()$ costing $O(1)$ operation (RAMAN; RAMAN; SATTI, 2007).

In order to decrease space usage and improve query performance, CAS encodes the sequence S using the wavelet tree (GROSSI; GUPTA; VITTER, 2003), a succinct self-indexed structure that stores a sequence of n symbols from an alphabet Σ of size σ using $O(n \log \sigma)$ bits. The standard version of wavelet trees uses a static Σ that cannot change after its creation.

This data structure extends bitvector’s $\text{rank}()$ and $\text{select}()$ to support also queries such as $\text{rank}_\alpha(S, i)$, $\text{select}_\alpha(S, f)$, $\text{range_count}_{[\alpha, \beta]}(S, i, j)$, $\text{range_next_value}_\alpha(S, i, j)$, and $\text{range_next_value_pos}_\alpha(S, i, j)$ with $O(\log \sigma)$ time complexity. Note that are alternative implementations in which using more space results in reducing time spent in some of these operations (NAVARRO, 2016).

The operation $\text{rank}_\alpha(S, i)$ retrieves the frequency of symbol α in the first i elements of the sequence S ; $\text{select}_\alpha(S, f)$ returns the position i such that the frequency of symbol α in s_1, s_2, \dots, s_i is f ; $\text{range_count}_{[\alpha, \beta]}(S, i, j)$ computes the number of symbols in $S = s_i, s_{i+1}, \dots, s_j$ considering only symbols in the interval $[\alpha, \beta] \subseteq \Sigma$; and $\text{range_next_value}_\alpha(S, i, j)$ gets the smallest symbol in s_i, s_{i+1}, \dots, s_j larger than α . Operation $\text{range_next_value_pos}_\alpha(S, i, j)$ returns the position of the symbol returned by

¹ An implementation of RRR is found in <<https://github.com/fclaude/libcds/blob/master/src/static/bitsequence/BitSequenceRRR.cpp>>. Accessed in April, 1st 2020.

$range_next_value_\alpha(S, i, j)$.

The wavelet tree also has the operation $range_report_{[\alpha, \beta]}(S, i, j)$, which retrieves separately the frequency of each distinct symbol in $S = s_i, s_{i+1}, \dots, s_j$ considering only symbols in $[\alpha, \beta] \subseteq \Sigma$ (GAGIE; NAVARRO; PUGLISI, 2012) with time complexity $O(k \log \sigma)$, where k is the number of symbols in the result.

CAS uses the wavelet tree by translating queries on the temporal graph represented by sequence S into wavelet tree queries on bitmaps. Rank and select queries on bitmaps have the same semantics of the equivalent queries on sequences and can be implemented with time complexity $O(1)$ and $o(\log n)$, respectively, where n is the length of B , with little impact on space (JACOBSON, 1988). There are others ways to implement these queries where the tradeoff time-space complexity can be changed.

A.3.1 Operation `has_edge`

In CAS, an algorithm to answer `has_edge(u, v, t1, t2)` needs to find the frequency of v in the block associated with u in S considering events in the interval $[t_{begin}, t_{end}]$. First, this algorithm finds the beginning and ending positions, i and j , of the block associated with the source vertex u in S .

Given that there is a bit set in B for each vertex in \mathcal{G} , running operation $select_1(B, u)$ returns us the starting position of the u -th block in B . Then, when it runs $i = rank_0(B, select_1(B, u))$ the returned value i points to the position of u block in S .

Similarly, to get the ending position j of the list of adjacencies of u in S , the algorithm first executes operation $select_1(B, u + 1)$ to get the position of the next symbol to u in B . The number of symbols 0 up to position $select_1(B, u + 1)$ in B is $rank_0(B, select_1(B, u + 1))$, which returns a value pointing one position after the end of the event list of u in S . Finally, the position j is $rank_0(B, select_1(B, u + 1)) - 1$.

Next, to know the number of events before t_{begin} and verify if u is active when starts the target interval, the algorithm calls $k_{begin} = range_next_value_pos_{t_{begin}}(S, i, j)$ to find the first position k_{begin} that has a timestamp symbol greater than t_{begin} to restrict S to the smaller block $S[i, k_{begin} - 1]$, which has only events that occurred until time t_{begin} .

Also, it calls $k_{end} = range_next_value_pos_{t_{end}}(S, i, j)$ to find the first position k_{end} that has a timestamp symbol greater than t_{end} to restrict S to the block $S[k_{begin}, k_{end} - 1]$, which allows to count the number of events that occurred during the interval $[t_{begin}, t_{end}]$.

Finally, the algorithm calls $range_count_{[v, v]}(S, i, k_{begin} - 1)$ to count the frequency f_{begin} of symbols v in $S[i, k_{begin} - 1]$ and $range_count_{[v, v]}(S, i, k_{end} - 1)$ to count the frequency f_{end} of symbols v in $S[k_{begin}, k_{end} - 1]$. If f_{begin} is odd, then edge (u, v) is active at time t_{begin} , else if f_{end} is greater than zero, then (u, v) is activated at some timestamp during

the interval $[t_{begin}, t_{end}]$, otherwise (u, v) is not active during $[t_{begin}, t_{end}]$. Note that, if it is a point-based query, where $t = t_{begin} = t_{end}$ and $f = f_{begin} = f_{end}$, then the algorithm only needs to check if f is odd to answer whether (u, v) is active or not at timestamp t .

A.3.2 Operation neighbors

An algorithm to answer $\text{neighbors}(u, t_1, t_2)$ needs to get the frequency of all possible target vertices v' in the block associated with u in S considering events from t_{begin} to t_{end} . Similar to $\text{has_edge}(u, v, t_1, t_2)$, it first finds the beginning and ending positions, i and j , respectively, of the block associated with u .

Then, it finds the position k_{begin} of the first symbol t_{begin} by calling $k_{begin} = \text{range_next_value_pos}_{t_{begin}}(S, i, k_{end})$ and the position k_{end} of the first symbol with value greater t_{end} by calling $k_{end} = \text{range_next_value_pos}_{t_{end}}(S, i, j)$ to restrict S to the block $S[k_{begin}, k_{end} - 1]$.

Next, the algorithm calls $\text{range_report}_{[\alpha, \beta] \subseteq V}(S, 1, k_{begin})$ to collect the frequency of all possible target vertices v' inside $S[1, k_{begin}]$ into C_1 and $\text{range_report}_{[\alpha, \beta] \subseteq V}(S, k_{begin} + 1, k_{end} - 1)$ to collect the frequency of all possible target vertices inside $S[k_{begin} + 1, k_{end} - 1]$ into C_2 .

Finally, it checks weak or strong semantics for every vertex collected. If the interval query has weak semantic, the algorithm only retrieves contacts in which edge (u, v') enters active in $[t_{begin}, t_{end}]$ — same as having odd frequency in C_1 — or whether it deactivates during $[t_{begin}, t_{end}]$ — same as having frequency greater than 0 in C_2 .

Instead, if the interval query has strong semantic, a contact is returned if an edge (u, v') enters active — same as having odd frequency in C_1 — and does not deactivate until t_{end} — same as having frequency equals to 0 in C_2 .

A.3.3 Operation neighbors^r

As the previous data structures, CAS also does not tackle $\text{neighbors}^r(u, t_1, t_2)$ efficiently. There are two different approaches to answer this query. In the first approach, an algorithm checks the frequency of symbols v in all blocks of S associated with source vertices u' . In order to determine these frequencies, first the algorithm restricts the symbols of each block, similarly to the previous algorithms. Next it calls $\text{range_count}_{[v, v] \subseteq V}$ inside each one, and, finally, it collects contacts in which an edge (u', v) holds weak or strong semantics during $[t_{begin}, t_{end}]$. A second approach, similar to previous structures, keeps another CAS structure where every edge would have its direction reversed. Then, it answers $\text{neighbors}^r(u, t_1, t_2)$ by calling $\text{neighbors}(u, t_1, t_2)$ in this second structure. The first approach would access all structure, which would impact severely in time, while the second approach would double the space required.

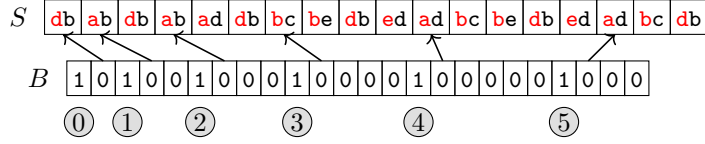


Figure 22 – CET representation of the temporal graph shown in Figure 2. CET structure has a sequence $S^{(2)}$ containing 2-dimensional symbols and a bitmap B . The t -th bit set to 1 in B marks the beginning of the $S^{(2)}$ block associated with the t -th timestamp. Each block associated with the t -th timestamp has symbols $(u, v) \in V \times V$ that represents an event of activation or deactivation of edge (u, v) at timestamp t . The arrows illustrate the beginning of each t -th block in $S^{(2)}$ and the first corresponding 0 in B .

A.4 Compressed Events Ordered by Time

Compressed Events ordered by Time (CET) uses a bi-dimensional sequence $S^{(2)} = s_1^{(2)}, s_2^{(2)}, \dots$, where symbols $s_i^{(2)}$ represent tuples $(u, v) \in V \times V$, and a bitmap B with size $|B| = |S^{(2)}| + |T|$ to mark timestamps (CARO; RODRÍGUEZ; BRISABOIA, 2015). As illustrated in Figure 22, it groups symbols in $S^{(2)}$ by time instead of grouping by source vertex. Each symbol $s_i^{(2)} = (u, v)$ in a block associated with time t represents an event of activation or deactivation of edge (u, v) at time t . Bitmap B marks the beginning of events associated with time t .

CET uses the interleaved wavelet tree data structure to store the sequence $S^{(2)}$ efficiently (CARO; RODRÍGUEZ; BRISABOIA, 2015). This structure generalizes the operations supported by standard wavelet trees to multidimensional sequences $S^{(d)} = s_1^{(d)}, s_2^{(d)}, \dots$, where d is the dimensionality of symbols $s^{(d)} = (s_1, s_2, \dots, s_d)$.

CET uses this structure to store edges (u, v) efficiently as bi-dimensional sequences. Then, it can be used later to retrieve graph information, such as edges frequency, by using queries such as $rank_{(u,v)}(S^{(2)}, i)$, $range_count_{\Sigma^{(2)}}(S^{(2)}, i, j)$, $range_report_{\Sigma^{(2)}}(S^{(2)}, i, j)$, $range_next_value_{(u,v)}(S^{(2)}, i, j)$, and $range_next_value_pos_{(u,v)}(S^{(2)}, i, j)$ in $O(\log \Sigma^{(2)})$ time complexity, where $\Sigma^{(2)}$ is the alphabet formed by every pair of vertices $V \times V$.

A.4.1 Operation has_edge

An algorithm to answer $has_edge(u, v, t_1, t_2)$ first finds the positions i and j in $S^{(2)}$ associated with $t_{begin} + 1$ and $t_{end} + 1$, respectively, by using the operation $select()$ of the bitvector implementing the bitmap B .

Then, it calls $rank_{(u,v)}(S^{(2)}, i - 1)$ to retrieve the frequency of events regarding edge (u, v) until timestamp t_{begin} and $rank_{(u,v)}(S^{(2)}, j - 1)$ to retrieve the frequency of events during the $[t_{begin}, t_{end}]$. Next, similar to previous strategies, it uses the parity property to answer whether edge (u, v) is active or not during the interval $[t_{begin}, t_{end}]$.

A.4.2 Operation neighbors

In order to answer $\text{neighbors}(u, t_1, t_2)$, an algorithm finds positions i and j associated to t and $t + 1$, then it performs $\text{range_report}_{(u,v) \subseteq V}(S^{(2)}, 1, i - 1)$. For weak semantic, if the frequency of symbol (u, v) is odd, it adds edge (u, v) to the result, then, for the remaining symbols, it calls $\text{range_report}_{(u,v) \subseteq V}(S^{(2)}, i, j - 1)$ and, if the frequency of symbol (u', v') is greater than 0, it also adds edge (u', v') to the result. For strong semantic, if the frequency of symbol (u, v) is odd, it discards edge (u, v) from the result, then, for the remaining symbols, it calls $\text{range_report}_{(u,v) \subseteq V}(S^{(2)}, i, j - 1)$ and removes from result the edges in which the corresponding symbols have frequency equal to 0.

A.4.3 Operation neighbors^r

Differently from the previous structures, CET has the same time complexity for retrieving direct and reverse neighbors of a given vertex v . An algorithm to answer $\text{neighbors}^r(u, t_1, t_2)$ is similar to $\text{neighbors}(u, t_1, t_2)$. The only difference is that it calls $\text{range_report}_{(u,v) \subseteq V}$ instead of $\text{range_report}_{(v,u) \subseteq V}$. In other words, we just swap the dimension values of symbols in $S^{(2)}$.

A.5 Temporal Graph Compressed Suffix Array

Temporal Graph Compressed Suffix Array (TGCSA) is a technique based on the Compressed Suffix Array (CSA) (SADAKANE, 2003) to store and query temporal graphs (BRISABOA et al., 2018). It represents a list of contacts using a string with unique characteristics and transforms the problem of querying temporal graphs in a substring matching problem. Therefore, TGCSA represents a list of contacts $C = c_1, c_2, \dots, c_n$, where $c_i = \{u, v, t_{begin}, t_{end}\} \in \mathcal{G}$, as a sequence $S = s_1, s_2, \dots, s_m$ formed by the concatenation of all c_i .

Note that, each element in a contact should be represented by a unique symbol. For this, TGCSA constructs and stores a dictionary Σ to encode S considering the following rules: $u \in [1, |V|]$, $v \in [|V| + 1, 2|V|]$, $t_{begin} \in [2|V| + 1, 2|V| + |T|]$ and $t_{end} \in [2|V| + |T| + 1, 2|V| + 2|T|]$. That is, symbols that encode source vertex have lower values than symbols that encode target vertex, which in turn, have lower values than activation timestamps and, which in turn, have lower values than deactivation timestamps. By using these rules, TGCSA can order the symbols in Σ in 4 different groups and take advantage of this property for speeding-up queries later. For now on, we will assume the string E to be the sequence of encoded contacts obtained from S by using codes in Σ . Note that, it is possible to decode symbols in E using Σ .

The standard Suffix Array (SA) strategy (MANBER; MYERS, 1993), depicted in

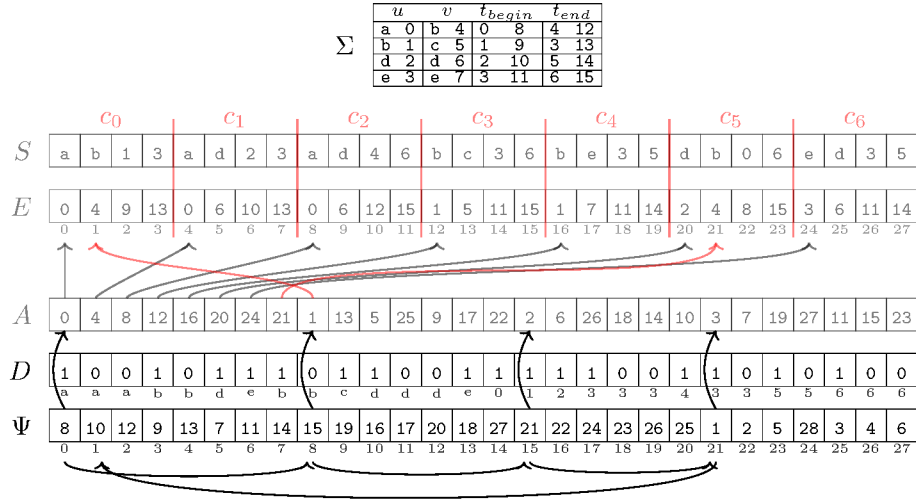


Figure 23 – TGCSA representation of the temporal graph shown in Figure 2. Temporal graph representation based on the uncompressed suffix array is represented by the structures with light gray color: the sequences S and E , and the suffix array A . Sequence S is formed by concatenating all contacts of the temporal graph. Sequence E has the encodings of symbols in S using the dictionary Σ , and A is obtained by ordering lexicographically the suffixes in E . TGCSA stores only some structures: a dictionary Σ , a bit array D , and a successor suffix array Ψ . D keeps track of A symbols by setting 1 at positions that symbols first appear in A , Ψ preserves the property $A[\Psi[i]] = A[i] + 1$, thus it is possible to retrieve the next symbol in the original sequence, and Σ is used to map original and encoded symbols. The arrows represent association examples between two different arrays or, in the case of Ψ , the application of successively calling itself to find the next symbols of a contact.

light gray in Figure 23, enumerates the suffixes from the string E and sorts them into an array of integers A . This approach allows finding substrings in E that match a given query in logarithmic time on the length of E by first encoding the substring query using Σ and, then, performing binary search in A to retrieve the possible matches. Moreover, this strategy can also retrieve the context of a match in constant time by accessing the surrounding symbols of E .

However, E and A can consume much space and, thus, in cases in which space is important, the CSA strategy is more appropriate. As show in darker color in Figure 23, instead of storing E and A , CSA stores a bitvector D in the format $10^{f_1}10^{f_2} \dots 10^{f_{|\Sigma|}}$, where f_i is the frequency of the i -th symbol in Σ , and the successor array Ψ with the property $A[\Psi[i]] = A[i] + 1$ to keep track of the next suffixes and, thus, the next positions of E symbols. Array Ψ can be further compressed by applying DeltaGap to each *run*, i.e. each already sorted part.

In order to find matches using the CSA data structure given a query substring $Q = q_1, q_2, \dots, q_l$, an algorithm first encodes Q into the substring $E' = e'_1, e'_2, \dots, e'_l$ using

Σ , then it calls $i = \text{select}_1(D, e'_1)$ and $j = \text{select}_1(D, e'_1 + 1) - 1$ to find the range of positions $[i, j]$ containing suffix candidates beginning with the symbol e'_1 . Next, for each suffix candidate Su with starting position at $k \in [i, j]$, it tries to retrieve its next symbol position by calling $k = \Psi[k]$ and, then, its next symbol value by calling $\text{rank}_1(D, k - 1)$. Next symbols for a candidate Su are successively retrieved until the algorithm finds a symbol that does not match the next symbol in the query E' or it successfully matches its first l symbols, *i.e.* the query size. In the first case, it discards safely the corresponding suffix candidate and, in the second, it returns the corresponding matching suffix Su .

A.5.1 Operation `has_edge`

An algorithm to answer `has_edge`(u, v, t_1, t_2) is similar to the string matching strategy we described. First, it encodes u to its corresponding encoding e' using the dictionary Σ . Then, it calls $i = \text{select}_1(D, e')$ and $j = \text{select}_1(D, e' + 1) - 1$ to find the beginning and ending positions of suffixes that start with the encoded symbol e' . Next, for each suffix candidate Su with starting position at $k = [i, j]$, it tries to construct a candidate contact $c = \{u', v', t'_{begin}, t'_{end}\}$, where $u' = u$, $v' = M(\text{rank}_1(D, \Psi[k]) - 1)$, $t'_{begin} = M(\text{rank}_1(D, \Psi[\Psi[k]]) - 1)$ and $t'_{end} = M(\text{rank}_1(D, \Psi[\Psi[\Psi[k]]]) - 1)$, where the function $M(\cdot)$ decodes symbols using Σ . Note that, $k = \Psi[k]$ retrieves the position of the next encoded symbol, $\text{rank}_1(D, k) - 1$ computes the encoded symbol at position k and $M(\cdot)$ decodes it back to the corresponding contact element. Finally, it collects all candidate contacts that intervals overlaps with $[t_{begin}, t_{end}]$. As the general substring matching case we presented earlier, the algorithm can also discard candidates that does not match the query immediately as next symbols are discovered.

A.5.2 Operation `neighbors`

An algorithm to answer `neighbors`(u, t_1, t_2), similarly, encodes u to its corresponding encoding e' using the dictionary Σ , finds the suffixes that start with the encoded symbol e' and constructs the candidate contacts. However, it collects only the candidate contacts that satisfy the required interval semantic. If `neighbors`(u, t_1, t_2) has weak semantic, then the algorithm collects the contacts in which $t'_{begin} \leq t_{end}$ and $t'_{end} \geq t_{begin}$, otherwise, if it has strong semantic, then it collects the contacts in which $t'_{begin} \leq t_{begin} \leq t_{end} \leq t'_{end}$.

A.5.3 Operation `neighborsf`

In order to answer `neighborsf`(u, t_1, t_2) efficiently, the authors introduced a modified version of the Ψ array. In the original array Ψ , if a suffix candidate Su with starting position at k starts with a symbol that corresponds to a deactivation timestamp encoding, then, the next suffix symbol at position $k = \Psi[k]$ would correspond

to the source vertex encoding of the next contact. In Brisaboa et al. (2018), the authors modified Ψ to make it cyclical regarding the same contact, thus, the suffixes in the last 25% positions of Ψ — those corresponding to deactivation time encodings — points to the suffixes corresponding to the source vertex encoding of the same contact. Hence, an algorithm to answer $\text{neighbors}^r(u, t_1, t_2)$, first encodes v and, then, it iterates the suffixes that starts with the v encoding to construct the candidate contacts $c = \{u', v', t'_{begin}, t'_{end}\}$, where, in this case, $u' = M(\text{rank}_1(D, \Psi[\Psi[\Psi[i]]]) - 1)$, $v' = v$, $t'_{begin} = M(\text{rank}_1(D, \Psi[i]) - 1)$ and $t'_{end} = M(\text{rank}_1(D, \Psi[\Psi[i]]) - 1)$. The next steps are the same as in $\text{neighbors}(\mathcal{G}, u, t_{begin}, t_{end})$. Therefore, $\text{neighbors}(u, t_1, t_2)$ and $\text{neighbors}^r(u, t_1, t_2)$ have the same time complexity.

A.6 Compressed k^d Tree

Compressed k^d tree (ck^d -tree) (CARO et al., 2016) is a compressed version of the k^d -tree that stores d -dimensional bitmaps efficiently (BERNARDO, 2014). As shown in Figure 24, the ck^d -tree represents recursively the decomposition of a d -dimensional bitmap into equal sized partitions. At each level, it splits the current bitmap partitions of size (s_1, s_2, \dots, s_d) into k^d smaller partitions of size $(\frac{s_1}{k}, \frac{s_2}{k}, \dots, \frac{s_d}{k})$ and redirects them to the lower level nodes. Each node stores a 1-dimensional bitmap B of size k^d to describe which partition can be further split. If the current partition only contains bits 0 then there is no need to split it further and, therefore, the corresponding position at bitmap B is set to 0. Otherwise, if it contains some bit 1 then the corresponding position at bitmap B is set to 1 and, additionally, the node holds a pointer to the next child that will split it further. To check the state of some bit b in a d -dimensional bitmap at position $p = (p_1, p_2, \dots, p_d)$, an algorithm traverses the tree following a top-down approach and, at each level, it search for the i -th partition that contains p and descend to the corresponding child whether $B_i = 1$. If at some point $B_i = 0$ then $b = 0$, otherwise, if the algorithm reaches an external node with $B_i = 1$ then $b = 1$.

Caro et al. (2016) stores and query temporal graphs by using ck^4 -trees with each dimension representing, respectively, source vertices, target vertices, activation times and deactivation times of contacts. As temporal graphs are usually sparse, a naïve implementation of the ck^4 -tree structure would traverse many nodes with only one child until it reaches an external node. In order to decrease the number of nodes with only one child and, consequently, improve space and query efficiency, the ck^4 -tree uses a second type of external node to store only the relative coordinates of the cell that has the single value 1 inside the current partition. Therefore, internal nodes also must store an additional 1-dimensional bitmap to differentiate the type of external nodes, with values being 0 if the corresponding children represent partitions with more than one bit 1 or 1 if children represent partitions with a single bit set to 1.

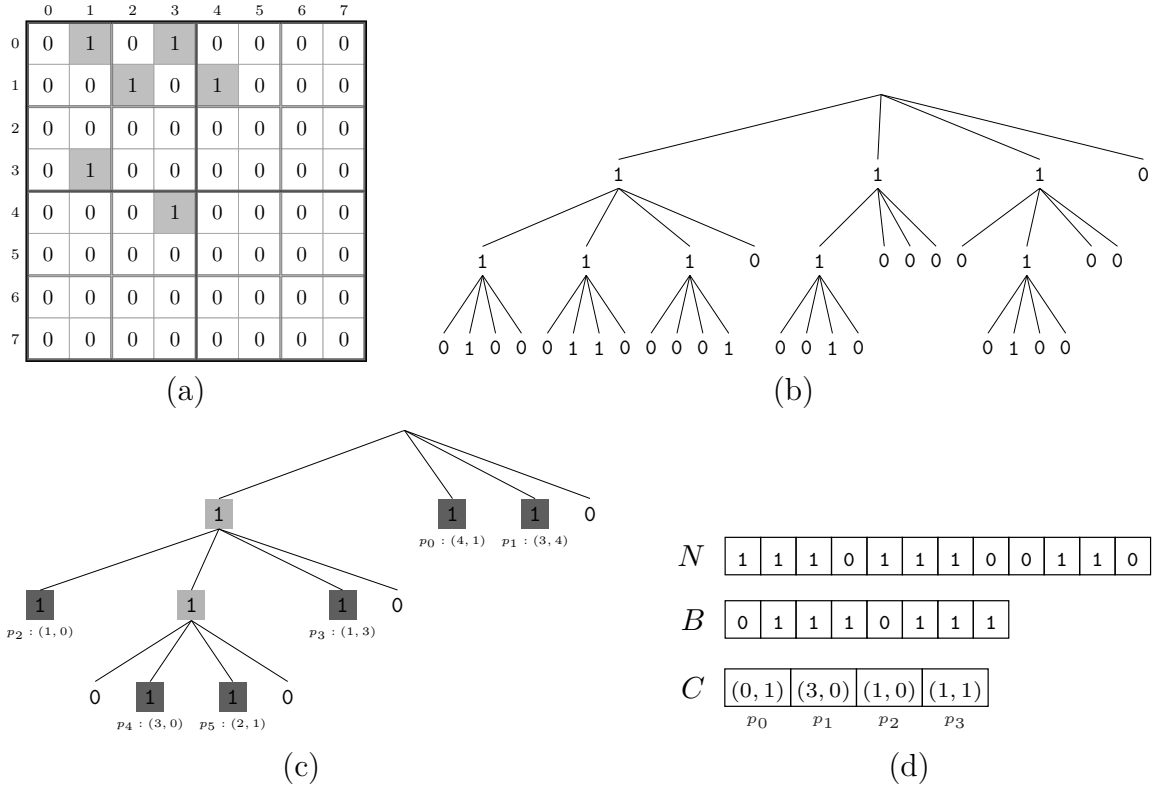


Figure 24 – k^d tree representation, with $k = 2$ and $d = 2$, of the underlying non-temporal graph shown in Figure 2 (only edges without timestamps). In (a), we show the 2-dimensional matrix that stores the binary relations between source and target vertices, u and v , respectively. In (b), we show the corresponding k^d tree without path compression. Each node has k^d children representing k^d different partitions of size $m/2^{l-1}$ inside the original matrix, where l is the level of a node in the tree. In (c), we show the corresponding ck^d tree, which compresses whole paths into black nodes. Instead of storing whole paths that results in single bits set to 1, the ck^d tree stores only the global coordinates regarding these single bits in a different type of node. Finally, in (d), we show the memory layout of the ck^d tree that contains a bitarray N , with information about bits set, a bitarray B , with information about colors for bits set to 1 and an array C of relative coordinates from the current partitions. Note that C does not store global coordinates to save space. Also, it does not store positions of black nodes in the last level since these positions can be obtained during searches.

A.6.1 Operation `has_edge`

An algorithm to answer `has_edge`(u, v, t_1, t_2) uses the `point`(\mathcal{T}, P) query internally, passing the ck^4 -tree \mathcal{T} constructed from \mathcal{G} and the d -dimensional point of interest $P = \langle u, v, t_{begin}, t_{end} \rangle$. Starting at the root node, the `point`(\mathcal{T}, P) query algorithm descends the ck^d -tree recursively checking at every level if there is a child partition that may contain P . If the algorithm does not reach an external node then there is no edge (u, v) active during the interval $[t_{begin}, t_{end}]$. Otherwise, depending on the type of the external node, it verifies whether the cell that contains P in the current partition is set to 1 to check if an edge (u, v) is active during the interval $[t_{begin}, t_{end}]$ or not. If the external node has an 1-dimensional bitmap B (type 1), the algorithm calculates the relative position P_r in the current partition using the path it traversed to reach this node and, then, it checks whether $B_i = 1$, where i is the position associated with the cell P_r in the current partition. Otherwise, if the node stores only the relative position C_r of the single bit 1 inside the corresponding partition (type 2), the algorithm calculates P_r and, then, it checks whether $P_r = C_r$.

A.6.2 Operation `neighbors`

An algorithm to answer `neighbors`(u, t_1, t_2) uses the `range`(\mathcal{T}, R) query internally by passing the ck^4 -tree \mathcal{T} constructed from \mathcal{G} and the d -dimensional region of interest R formed by the lower boundary $L = \langle u, \min(V), \min(T), t_{begin} + 1 \rangle$ and the upper boundary $U = \langle u, \max(V), t_{begin}, \max(T) \rangle$. Starting at the root node, the `range`(\mathcal{T}, U) query algorithm descends recursively all children whose partitions overlap R until it reaches all possible external nodes. Then, for each external node, the algorithm computes the bit 1 coordinates that overlap R depending on its type and adds them to the result set. If the external node has a 1-dimensional bitmap B (type 1), the algorithm searches for the positions i where $B_i = 1$, compute the relative positions associated with positions i and calculate the global coordinates based on the path it traversed to reach the node. Otherwise, if the node stores only the relative position of the single bit 1 inside the corresponding partition (type 2), the algorithm simply calculates the global coordinate based on the path it traversed.

A.6.3 Operation `neighborsr`

An algorithm to process `neighborsr`(u, t_1, t_2) is similar to the direct query. However, it fixes the second dimension — the dimension associated with target vertices — instead of the first when calling `range`(\mathcal{T}, R), thus, the region R is formed by $L = (\min(V), v, \min(T), t_{begin} + 1)$ and $U = (\max(V), v, t_{begin}, \max(T))$. As CET, finding direct and inverse neighbors using TGCSA have the same time complexity.

Table 4 – Worst-case space cost of the temporal graph representations using the number of vertices n , number of edges m , number of contacts c and the lifetime t of \mathcal{G} .

Representation	Worst-case Space
EveLog	$O(c \log \frac{nt}{c} + n \log (n + c))$
EdgeLog	$O(m \log \frac{nc}{m} + c \log \frac{mt}{c} + n \log m)$
CAS	$O(c \log (n + t) + n)$
CET	$O(c \log m + t)$
TGCSA	$O(c \log (n + t))$
ck^d tree ($d = 4$)	$O(c \log \frac{nt}{c})$

- Cost of pointers was considered when necessary

A.7 Considerations

In this section we compare the data structures we review based on their worst-case space cost and time cost for answering some queries we described in Chapter 2. In order to get the information of costs, we simplified the expanded formula, when available, in the original using the big- O notation. The considered variables are: n , for the number of vertices; m , for the number of edges in the underlying static graph; c , for the number of contacts; and t , the for lifetime of the temporal graph. For a more detailed description see the work by (CARO; RODRÍGUEZ; BRISABOA, 2015).

Table 4 shows the space cost of the data structures we reviewed. In this comparison, we considered the cost of pointers when necessary. For instance, the structures EveLog and EdgeLog stores pointers to map source vertices and their corresponding temporal adjacency lists or event lists. The main sources of space consuming for the EveLog structure are the number of contacts and vertices. This is due to the number of items in the temporal adjacency lists, c , and the number of pointers mapping vertices to their corresponding temporal adjacency lists, n .

In the case of EdgeLog, there are three major sources of space usage: number of edges, number of contacts and number of vertices. This is because EdgeLog extends adjacency list for temporal graphs, however, additionally, it stores pointers for time intervals for every edge in the underlying edge, it stores pointers for every vertex to their corresponding temporal adjacency list, and it stores an additional list of time intervals.

For CAS the major sources of space consumption are the amount of contacts and the amount of vertices. This is due the size of the sequence to store edge activation and deactivation events that depends c and stores $n + t$ symbols. Additionally, there is also a bitvector to store positions in which events corresponding to a source vertex begin in the sequence based on the amount of vertices n .

CET increases space consumption according to the amount of contacts and the lifetime of the temporal graph. Similarly to CAS, CET stores a sequence with size based

Table 5 – Time cost of `has_edge` and `neighbors` queries with a timestamp parameter using the number of vertices n , number of edges m , number of contacts c and the lifetime t of \mathcal{G} .

Structure	<code>has_edge</code>	<code>neighbors</code>
EveLog	$O(\frac{c}{n})$	$O(\frac{c+m}{n})$
EdgeLog	$O(\frac{c}{m} + \frac{m}{n} + \log(\frac{c}{n}))$	$O(\frac{c}{n} + \frac{m}{n} \log(\frac{c}{m}))$
CAS	$O(\log(n+t))$	$O(\frac{m}{n} \log(n+t))$
CET	$O(\log n)$	$O(\frac{m}{n} \log n)$
TGCSA	$O(\frac{c}{m} \log(c))$	$O(\frac{c}{n} \log(c))$
ck^d tree ($d = 4$)	$O(c^{\frac{1}{2}})$	$O(c^{\frac{3}{4}})$

- Uniform degree distribution in the aggregate graph was considered when necessary

 Table 6 – Time cost of `neighborsr` and `aggregate` queries with a timestamp parameter using the number of vertices n , number of edges m , number of contacts c and the lifetime t of \mathcal{G} .

Structure	<code>neighbors^r</code>	<code>aggregate</code>
EveLog	unfeasible	$O(c+m)$
EdgeLog	unfeasible	$O(c+m \log(\frac{c}{m}))$
CAS	$O(\frac{c+m}{n} \log(n+t))$	$O(m \log(n+t))$
CET	$O(\frac{m}{n} \log n)$	$O(m \log n)$
TGCSA	$O(\frac{c}{n} \log(c))$	$O(c \log c)$
ck^d tree ($d = 4$)	$O(c^{\frac{3}{4}})$	$O(c)$

- Uniform degree distribution in the aggregate graph was considered when necessary

on the amount of contacts, however, the number of symbols is based on the number of the underlying edges and the size of the bitvector is based on the variable t since it marks the start of each timestamp in the sequence.

The major source of space consumption of TGCSA is linked to the amount of contacts since it compresses the sequence containing the concatenation of all contacts and the suffix array of the same size containing the surroundings of each symbol in the original sequence. This is due that TGCSA store a constant amount of data structures with size depending on the amount of contacts. For example, it needs to store a dictionary that maps every symbol in contacts to their corresponding code, a bitvector that stores information about symbols in the ordered suffix array and a sequence that stores information about the surroundings of symbols in the original sequence.

In the case of ck^d tree, being $d = 4$ due to the dimension of contacts, the major source of space consumption is the number of contacts. This is because the ck^4 tree compresses a tensor of degree 4 containing c bits set with dimensions size based on the number of vertices n and the lifetime of the temporal graph t .

Tables 5 and 6 show the time cost of the data structures to answer some queries

we described in Chapter 2. In this table, we compare the following operations: `has_edge`, `neighbors`, `neighborsr` and `aggregate`. We note that there are only queries based on a single timestamp and for the costs, as the original authors, we considered a graph generated using uniform degree distribution. As we can see, for EveLog, the cost of the query `has_edge` depends on the average number of events of activation or deactivation in the temporal adjacency list associated with vertex u to find the events with vertex v until time t . Similarly, the cost of `neighbors` also depends on the average number of events of the source vertex u but it needs also to consider the average number of edges in the underlying graph in that u participates. The `neighborsr` is unfeasible using only the temporal adjacency lists for the outgoing contacts because it would traverse the entire structure. One can also store another structure that considers the incoming contacts of a vertex and get time costs similar with the `neighbors` query, however, it would double the space needed. Finally, for `aggregate`, EveLog needs to compute one `neighbors` for all vertices $u \in V$ to construct the snapshot \mathcal{G}_t .

For EdgeLog, the operation `has_edge` needs to decompress the temporal adjacency list associated with a source vertex u with average size $\frac{m}{n}$ and the list of time intervals associated with a destination vertex v with size $\frac{c}{m}$. Also, it needs to run binary searches to check if this edge is active at timestamp t using a binary search. For the operation `neighbors`, EdgeLog need to decompress the temporal adjacency list associated with vertex u and all lists of time intervals associated with vertex v . Then it needs to binary search the lists containing intervals to check if edge (u, v) is active at time t . For the same reasoning of EveLog, the operation `neighborsr` is unfeasible for EdgeLog and it also can spend about the double the space to have similar costs of `neighbors`. Finally, `aggregate` uses one `neighbors` for every vertex $u \in V$ and a binary search is performed for every edge (u, v) in the corresponding list of time intervals associated with vertex v . During this process, all the temporal graph ends up being decompressed.

Differently, the other data structures do not need a decompressing step. For CAS, the operation `has_edge` perform operations in the underlying wavelet matrix that stores a sequence representing temporal adjacency lists in for form of events of activation and deactivation. As this sequence has $n + t$ symbols, a query to count the amount of (v, t) occurrences in the block associated with vertex v at time t is $O(\log(n + t))$. For query `neighbors`, this same reasoning is made for every edge (u, v) . In this case, the average value per vertex is $\frac{m}{n}$ and, therefore, the total cost is $O(\frac{m}{n} \log(n + t))$. The `neighborsr` operation becomes feasible with the CAS structure because it does not need to decompress the whole structure as the other two structures. However, it still needs to call one wavelet tree operation in every block associated with some other vertex $u \in V$ to count the number of events (u, v) in time t . Finally, for `aggregate`, CAS calls the query `neighbors` for every vertex $u \in V$ and, thus, it needs to execute one wavelet matrix operation for each edge (u, v) for $u, v \in V \times V$.

For CET, the operation `has_edge` also uses the operation to count occurrences on the underlying wavelet matrix representing the sequence of events of activation or deactivation. However, different of CAS, it stores on the sequence pair of vertices or edges (u, v) and uses the additional bitvector to split the sequence in blocks with the same timestamp. Therefore, in the sequence there are only vertex symbols and, therefore, the cost to count the number of occurrences of (u, v) in the block associated with t is $O(\log n)$. For the operation `neighbors`, CET calls $\frac{m}{n}$ times, the average number of edges for vertex, the wavelet tree operation, thus the total cost is $O(\frac{m}{n} \log n)$. The algorithm for `neighborsr` is similar to `neighbors`, it only needs to change the symbols (u', v) being counted and, thus, CET is the first compact data structure to present the same costs for the both operations. Finally, for `aggregate`, similar to the other structures so far, it needs to call `neighbors` for all vertices $u \in V$ and, therefore, it counts occurrences for all m edges (u, v) with $v \in V$ as well.

For TGCSA, the operation `has_edge` spends $O(\log c)$ cost to convert symbols from the data structures to symbols in the original sequence of concatenated contacts. Therefore, after each call to the array Ψ , the TGCSA needs to decode the symbol using the dictionary to reason about the resulting symbol. As the substring uv can discard mostly candidates in the string matching algorithm, there are the average $O(\frac{c}{m})$ timestamps to be checked for edge u, v . For the `neighbors`, the algorithm cannot filter contact candidates in the string matching process as for `has_edge` using symbol v , therefore it calls $O(\frac{c}{n})$ times the access operation in the array Ψ and, thus, the same amount of operations for decoding the resulting symbol using the dictionary. Similarly to CET, the TGCSA structure also can answer the operation `neighborsr` with the same cost of the `neighbors` query. The reason is that the subsequence matching can be circular using the array Ψ and, thus, the process can start at symbol filtering candidates by the coded symbol for symbol t and then use u to continue the process. Finally, for `aggregate` TGCSA also needs to call n times the query `neighbors` for all vertices $u \in V$ and, therefore, all contact symbols must be accessed and decoded.

For the ck^d tree, all the graph queries are translated to a range query in the 4-dimensional tensor using a 4 – *dimensional* rectangle $R = (L, U)$ consisting of the lower and upper bound points L and U , respectively. For the operation `has_edge`, as there are 2 dimensions known, the algorithm constructs an rectangle $R = (L, U)$, where $L = \langle u, v, 0, t \rangle$ and $U = \langle u, v, t, \max(T) \rangle$. Note that the algorithm needs to search in the region where $t_{begin} \leq t$ and $t_{end} \geq t$ to find contacts with intervals that contains t . Therefore, at every level of the tree, the search algorithms can descend to k^2 children since half associated with the two first dimensions do not pass the test. Also, the height of the tree is $h = \log_{k^4} c = \frac{1}{4} \log_k c$ and, thus, the algorithm visits at most $(k^2)^h = O(c^{\frac{2}{4}})$ nodes. For the operation `neighbors`, there are only one known dimension and the algorithm constructs the rectangle consisting of $L = \langle u, 0, 0, t \rangle$ and $U = \langle u, \max(V), t, \max(T) \rangle$.

Therefore, the algorithm descends at most $(k^3)^h = O(c^{\frac{3}{4}})$ nodes since just k^1 nodes are not visited at every level. For the operation **neighbors**, the ck^d tree has the same cost as there is only one fixed dimension and the constructed rectangle consists of $L = \langle 0, v, 0, t \rangle$ and $U = \langle \max(V), v, t, \max(T) \rangle$. Finally, for the operation **aggregate** there is no fixed dimension. Therefore, it constructs a rectangle consisting of $L = \langle 0, 0, 0, t \rangle$ and $U = \langle \max(V), \max(V), t, \max(T) \rangle$ and descends all k^4 children every level and, thus, the cost is $(k^4)^h = O(c)$.

Join and split operations for B^+ -trees

In this chapter, we will present the preliminaries for comparing our novel data structure based on sequential arrays on disk with the approach introduced in (BRITO et al., 2022) using self-balanced Binary Search Trees (BSTs). For the latter, we used B^+ -trees (ABEL, 1984) as a replacement for the in-memory BSTs. First, we describe in Section B.1 an algorithm to perform the `join` operation on B^+ -trees. Then, we describe in Section B.2 an algorithm to perform the `split` operation on B^+ -trees. They used both operations in the original article to remove a range of redundant intervals in $O(\log(\tau))$ time.

Briefly, the approach introduced in (BRITO et al., 2022) stores, in a matrix $n \times n$, pointers to BSTs containing time intervals. In each BST, only non-redundant intervals are kept, *i.e.*, those that do not contain another interval in the same tree. In order to insert a new interval I into a tree T , their algorithm has three steps: (1) check whether I do not contain any other interval in T ; (2) remove any interval that becomes redundant, *i.e.*, those that contain I ; and (3) insert I into T . Many intervals can become redundant during (2); nevertheless, they appear as an ordered sequence of intervals.

The algorithm introduced in (BRITO et al., 2022) removes such sequence of redundant intervals using a `join` after `splits` approach. Let L_1 and L_2 be the endpoints of the interval sequence to be removed from a BST T . First, their algorithm splits T in two trees T_{left} and T_{mid} using `split`(T, L_1). Then, it splits T_{mid} in two other trees $T_{redundant}$ and T_{right} by calling `split`(T_{mid}, L_2). Finally, it merges T_{left} and T_{right} with the operation `join`(T_{left}, T_{right}).

Let each leaf node of B^+ -trees contains an array K of keys of size N and a pointer to its next sibling. Let each non-leaf node contains an array of keys K of size M and an additional array of pointers C to child nodes of size $M + 1$. Additionally, assume that every node contains the height of the sub-tree it belongs, a pointer to its leftmost leaf child and a pointer to its rightmost leaf child. We note that we use these additional per-node

data to simplify our algorithms and discussions. In a real implementation, only the root node (the tree itself) must maintain them during the insertion and update operations. Information regarding the rest of the nodes can be computed during the execution of the next algorithms without increasing complexities.

B.1 Join operation for B^+ -trees

Algorithm 10 performs the operation `join` for B^+ -trees. Given two B^+ -trees T_{left} and T_{right} , such that keys present in T_{left} are smaller to keys in T_{right} , it must merge both trees in order to create a new valid B^+ -tree T containing all keys present in T_{left} and T_{right} . As B^+ -trees place leaf nodes at the same level, it simply inserts or shares the data present in the root node of the smaller tree into the appropriate node at the same height in the bigger tree. Then it maintains the B^+ -tree invariances up to its root node of the changed bigger tree whenever necessary. First, in line 11, the algorithm sets the next sibling of the rightmost leaf of T_{left} to be the leftmost leaf of T_{right} . Then, if $\text{height}(T_{left}) \geq \text{height}(T_{right})$, in lines 13 and 14, it adds T_{right} to T_{left} by calling `JOINRIGHT(T_{left}, T_{right})` and returns T_{left} ; otherwise, in lines 16 and 17, it adds T_{left} to T_{right} by calling `JOINLEFT(T_{left}, T_{right})` and returns T_{right} . From lines 1 to 11, we detail the `JOINRIGHT` routine, the `JOINLEFT` routine is implemented symmetrically. In line 1, the algorithm descends T_{left} until reaching the rightmost node n_{left} at the same height of the root node of T_{right} . If a single node of size B can fit the content of both n_{left} and the root node of T_{right} , in line 5, it simply merges both nodes by adding to n_{left} the data present in the root node of T_{right} . Otherwise, in line 7, it equally shares the data of both nodes, and, in line 8, it inserts into the parent of n_{left} a new key together with a pointer to the root node of T_{right} . If the parent node has no space left to accommodate the new data, a node splitting routine must be invoked and this process can continue up to the root node of T_{left} . Finally, if the algorithm needs to split the current root node of T_{left} , then it creates a new root node, and, in this case, in line 10, it increments the height of T_{left} by one.

Theorem 15. *Algorithm 10 accesses $O(|\text{height}(T_{left}) - \text{height}(T_{right})|)$ pages in the worst-case.*

Proof. In line 11, the algorithm accesses one page to set the next child node of the rightmost leaf node of T_{left} . Then, it calls `JOINRIGHT` or `JOINLEFT` depending on the heights of T_{left} and T_{right} , both accessing the same amount of pages. Without loss of generality, assume that $\text{height}(T_{left}) \geq \text{height}(T_{right})$ and it calls thus the `JOINRIGHT` procedure. Then, at line 2, the algorithm accesses $\text{height}(T_{left}) - \text{height}(T_{right})$ pages while descending to the rightmost node of T_{left} at height $\text{height}(T_{right})$. Next, if there is enough room to fit the data of both nodes being merged in a single node, it accesses

Algorithm 10 $\text{join}(T_{\text{left}}, T_{\text{right}})$ **Require:** Two trees of intervals T_{left} and T_{right}

```

1: procedure JOINRIGHT( $T_{\text{left}}, T_{\text{right}}$ )
2:    $n_{\text{left}} \leftarrow \text{descend\_right}(T_{\text{left}}, \text{height}(T_{\text{left}}) - \text{height}(T_{\text{right}}))$ 
3:    $n_{\text{right}} \leftarrow \text{root}(T_{\text{right}})$ 
4:   if  $\text{size}(n_{\text{left}}) + \text{size}(n_{\text{right}}) \leq B$  then
5:      $n_{\text{left}} \leftarrow \text{merge}(n_{\text{left}}, n_{\text{right}})$ 
6:   else
7:      $\text{share}(n_{\text{left}}, n_{\text{right}})$ 
8:      $\text{insert\_rec}(\text{parent}(n_{\text{left}}), \text{min\_key}(n_{\text{right}}), n_{\text{right}})$ 
9:     if new root node was created then
10:       $\text{height}(T_{\text{left}}) \leftarrow \text{height}(T_{\text{left}}) + 1$ 
11:  $\text{next\_leaf}(\text{rightmost\_leaf}(T_{\text{left}})) \leftarrow \text{leftmost\_leaf}(T_{\text{right}})$ 
12: if  $\text{height}(T_{\text{left}}) \geq \text{height}(T_{\text{right}})$  then
13:   JOINRIGHT( $T_{\text{left}}, T_{\text{right}}$ )
14:   return  $T_{\text{left}}$ 
15: else
16:   JOINLEFT( $T_{\text{left}}, T_{\text{right}}$ )
17:   return  $T_{\text{right}}$ 

```

$O(1)$ pages and the algorithm ends. Otherwise, it accesses $O(1)$ pages to share the content present in the considered nodes. Then, it accesses, again, $O(\text{height}(T_{\text{left}}) - \text{height}(T_{\text{right}}))$ pages in order to insert new key and pointer pairs up to the root of T_{left} in the worst-case. Finally, if a new node is created, it access one more page to increment the height of T_{left} and the algorithm ends. \square

B.2 Split operation for B^+ -trees

Algorithm 11 performs the operation `split` for B^+ -trees. Given an interval key L , it must split a tree T in two trees T_{left} and T_{right} such that all keys in T_{left} are smaller than L and all keys in T_{right} are greater or equal to L . To accomplish this task, it recursively descends T from the root node to the leaf node containing the biggest key less than L while partitioning nodes appropriately and, during the backward phase of the recursion, progressively building T_{left} and T_{right} . During each recursive step, in line 1, the algorithm first finds the position k in the current root node such that $K[k] \geq L$, where $C[k]$ is the pointer that branches to the next child node for non-leaf nodes. If the current root node is a leaf, in line 3, it partitions the current node in two sub-trees: T_{left} , containing a node with $K[1 \dots k - 1]$; and T_{right} , containing a node with $K[k \dots N]$. Then, in line 4, it sets the next sibling of T_{left} 's root to nil ; in line 5, it sets the next sibling of T_{right} to the next sibling of T 's root; and, in line 6, it returns $(T_{\text{left}}, T_{\text{right}})$. Note that no other leaf node besides the affected ones must update the pointer to its next siblings since the resulting trees will reuse the previous linkages. Next, if the current node is a non-leaf, in line 7,

the algorithm partitions the current node in three sub-trees: T_{left} , containing a root node with $K[1 \dots k-2]$ and $C[1 \dots k-1]$; T_{child} , containing a root node with $K[k-1 \dots k]$ and $C[k]$; and (3) T_{right} , containing a root node with $K[k+1 \dots M]$ and $C[k+1 \dots M+1]$. Additionally, whenever a sub-tree have only one pointer in its root node, its respective root node becomes the child pointed by it in order to maintain the correct B^+ -tree layout. Then, in line 8, it calls the SPLIT algorithm itself passing T_{child} as parameter and obtaining two sub-trees T'_{left} and T'_{right} as the intermediate result. Finally, in line 9, it advances the intermediate result by joining them appropriately with the sub-trees of the current level.

Algorithm 11 $\text{split}(T, L)$

Require: A tree of intervals T and a key interval L

```

1:  $k \leftarrow \text{find\_key\_position}(\text{root}(T), L)$ 
2: if  $\text{root}(T)$  is a leaf then
3:    $(T_{left}, T_{right}) \leftarrow \text{split\_leaf}(\text{root}(T), k)$ 
4:    $\text{next\_leaf}(\text{root}(T_{left})) \leftarrow \text{nil}$ 
5:    $\text{next\_leaf}(\text{root}(T_{right})) \leftarrow \text{next\_leaf}(\text{root}(T))$ 
6:   return  $(T_{left}, T_{right})$ 
7:  $(T_{left}, T_{child}, T_{right}) \leftarrow \text{split\_non\_leaf}(\text{root}(T), k)$ 
8:  $(T'_{left}, T'_{right}) \leftarrow \text{split}(T_{child}, L)$ 
9: return  $(\text{join}(T_{left}, T'_{left}), \text{join}(T'_{right}, T_{right}))$ 

```

Theorem 16. *Algorithm 11 accesses $O(\log_B(\tau))$ pages in the worst-case where τ is the maximum number of keys in the tree.*

Proof. During each recursive step, Algorithm 11 needs to: (1) partition the current node being considered in at least two sub-trees; (2) change pointers to next leaf siblings, whether the current node is a leaf; and (3) join the current sub-trees with the sub-trees resulting from the next recursive step. For (1), the algorithm accesses $O(1)$ pages. In the worst-case scenario, if the root node of a sub-tree has only a single child, the algorithm reads this child in order to make it the new root node. For (2), the algorithm also accesses $O(1)$ pages since only two leaf nodes are updated. For (3), the algorithm calls the `join` algorithm twice. Without loss of generality, consider only calls maintaining T_{left} . From the node above the leaf node containing the split key up to the root of T , the algorithm joins the current left sub-tree T_{left} with the intermediate left sub-tree T'_{left} resulting from the previous iteration. At each iteration there is a `join`(T_{left}, T'_{left}) call in which T_{left} is either empty or non-empty. In case T_{left} is empty, the algorithm pays nothing and, at the next iteration, the difference in height between T_{left} and T'_{left} increases by one. In case T_{left} is non-empty, the algorithm pays the difference in height accumulated so far and, at the next iteration, the difference in height resets to one. Therefore, as the summation of all payments is at most the height of the tree, the algorithm accesses $O(\log_B(\tau))$ pages while processing all `join` calls. \square