WARWICK
THE UNIVERSITY OF WARWICK

A Thesis Submitted for the Degree of PhD at the University of Warwick

Permanent WRAP URL:

http://wrap.warwick.ac.uk/176730

Copyright and reuse:
This thesis is made available online and is protected by original copyright.
Please scroll down to view the document itself.
Please refer to the repository record for this item for information to help you to cite it.
Our policy information is available from the repository home page.

For more information, please contact the WRAP Team at: wrap@warwick.ac.uk

warwick.ac.uk/lib-publications

WARWICK
THE UNIVERSITY OF WARWICK

Hypergraph-based Optimisations for Scalable Graph Analytics and Learning

by

Aparajita Haldar

Thesis

Submitted to the University of Warwick

in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

Department of Computer Science

December 2022

Contents

List of Tables iii

List of Figures iv

List of Algorithms vi

Acknowledgements vii

Declarations viii

Abstract x

Abbreviations xi

Symbols xiii

Chapter 1 Introduction 1

1.1 Research challenges 4
1.2 Contributions 6
1.3 Thesis structure 8

Chapter 2 Background 10

2.1 Graph-structured data 10
2.2 Partitioning graph-structured data 14
2.3 Graph analytics tasks 16
2.4 GNN models 17
2.5 Graph processing systems 19

Chapter 3 Related Work 20

3.1 Graph algorithms 20
3.2 Distributed systems 22
3.3 Partitioning methods 25

Chapter 4 Hypergraph Sampling for Dynamic Analytics 29

4.1 Preliminaries 31
4.2 Temporal Independent Cascade (T-IC) model 32
4.3 Hypergraph-based reachable set construction 34

4.4 Optimisation objectives . . . 37
4.5 RSM and ESM solution algorithms . . . 38
4.6 Experimental evaluation . . . 41
4.7 Discussion . . . 50

Chapter 5 Hypergraph Partitioning for Static Analytics **52**
5.1 Preliminaries . . . 53
5.2 Partitioning models for scalable similarity search . . . 56
5.3 Technique for evaluation of solution quality . . . 59
5.4 Experimental evaluation . . . 60
5.5 Discussion . . . 65

Chapter 6 Hypergraph Partitioning for Static GNNs **67**
6.1 Preliminaries . . . 68
6.2 Data-parallel GCN training algorithm . . . 69
6.3 Partitioning models for full-batch training . . . 74
6.4 Partitioning model using hypergraph sampling for mini-batch training . . . 78
6.5 Experimental evaluation . . . 79
6.6 Discussion . . . 86

Chapter 7 Hypergraph Partitioning for Streaming GNNs **88**
7.1 Preliminaries . . . 89
7.2 Streaming partitioner design . . . 90
7.3 Experimental evaluation . . . 94
7.4 Discussion . . . 99

Chapter 8 Conclusions **101**
8.1 Discussion . . . 101
8.2 Limitations and future work . . . 104
8.3 Concluding remarks . . . 107

List of Tables

4.1 Dataset properties 43
4.2 Normalised performance (reverse spread and binary success rate) at $|T|=25$ with different sizes of solution set $|S|=k$ 45
4.3 Normalised performance (expected spread) at $|T|=25$ with different sizes of solution set $|S|=k$ 46
4.4 Running time (in seconds) with different number of nets $|\mathcal{N}|$ (for $|T|=5$) . . 50
4.5 Running time (in seconds) with different number of time windows $|T|$ (for $|\mathcal{N}|=20K$) 50
4.6 Running time (in seconds) with different sizes of solution set $|S|=k$ (for $|T|=3$) on `Haslemere` 50

5.1 Dataset properties 61

6.1 Dataset properties 79
6.2 Performance comparison with HP, GP, and RP on $p=512$ CPUs 81
6.3 Performance comparison with HP and RP ($d = 1, 2, 5$) on `ogbn-Papers100M` for $p = 27$ GPUs 86
6.4 Running time (in seconds per epoch) compared to SOTA on `reddit` 87

7.1 Dataset properties 95

8.1 Summary of thesis contributions 102

List of Figures

1.1 Illustrative pipeline 4
1.2 Thesis structure (chapters) and contributions with corresponding research questions (RQs) 9

2.1 Hypergraph-based sampling and hypergraph transformation techniques . . 13

4.1 Example of dynamic propagation probabilities in the network 32
4.2 RSM vs ESM on g_{ij}^{T} 41
4.3 RSM vs ESM on g_{ij} 41
4.4 Normalised reverse spread with different lengths of time window $|T|$ 47
4.5 Normalised binary success rate with different lengths of time window $|T|$ . . 48
4.6 Normalised expected spread with different lengths of time window $|T|$ . . . 49

5.1 Approximate RoleSim* node-pair similarity retrieval performance on graph $\mathcal{G}$ 57
5.2 Approximate RoleSim* node-pair similarity retrieval 58
5.3 Performance comparison with graph-based partitioning on `DBLP` 62
5.4 Performance comparison with hypergraph-based partitioning on `DBLP` . . . 62
5.5 Performance comparison with graph-based partitioning on `AMZ` 63
5.6 Performance comparison with hypergraph-based partitioning on `AMZ` 63
5.7 Effect of sampling ratio η and weight μ on ranking quality on `DBLP` 64
5.8 Effect of sampling ratio η and weight μ on ranking quality on `AMZ` 64

6.1 Communication and computation processes during feedforward and back-propagation phases of the distributed GCN algorithm 73
6.2 Hypergraph partitioning of graph $\mathcal{G}$ 77
6.3 Strong scaling for full-batch training with HP, GP, RP, and CAGNET (CN) . 82
6.4 Performance comparisons for full-batch training with HP, GP, RP, and CAGNET (CN) 84
6.5 Performance comparisons for full-batch training. Speedup with increasing layers ($L = 3, 4, \ldots, 8$) and dimensions ($d = 50, 100$) on `roadNet-CA` for $p = 512$ CPUs 85
6.6 GNN model accuracy with HP on `Cora` for $p = 1$ to $p = 27$ GPUs 85
6.7 Performance comparisons for mini-batch training with HP and SHP 86

7.1 Streaming partitioner functionality within the dataflow pipeline 91

7.2 Performance comparison with GNN and HGNN on graph and hypergraph formats of `tags-ask-ubuntu` 97
7.3 Performance comparison with GNN and HGNN on graph and hypergraph formats of `threads-math-sx` 97
7.4 Performance comparison with different partitioners on `reddit-hyperlink` 98

List of Algorithms

4.1 RSM Solution . . . 39
4.2 ESM Solution . . . 40

6.1 GCN Parallel Feedforward . . . 71
6.2 GCN Parallel Backpropagation . . . 72
6.3 Stochastic Hypergraph Partitioning . . . 78

7.1 Streaming Partitioner . . . 92
7.2 Physical Part Computation . . . 93

Acknowledgements

I am deeply grateful for the progressive environment at Warwick and for my generous funding through the Feuer Scholarship.

More than anyone else, I thank my supervisor Prof. Hakan Ferhatosmanoglu, who has been a wonderful mentor. Our brainstorming sessions for new project ideas will always remain some of my cherished memories from these past years, and I feel fortunate to have experienced Hakan's infectious enthusiasm and curiosity. His meticulous guidance, as simultaneously my strongest critic and most ardent supporter, helped me to grow immeasurably as a researcher.

My proclivity for collaboration was recognised and nurtured through several productive partnerships at Warwick and beyond. I thank Gunduz Demirci, Rustam Guliyev, Shuang Wang, Sima Iranmanesh, and all my other co-authors who inspired me with their hard work.

I am grateful to Prof. Sunil Prabhakar and Prof. Yulan He for being my thesis examiners and providing valuable constructive feedback, and to Prof. Mike Joy for being my examination advisor. The inputs from Yulan and Prof. Stephen Jarvis as my advisors, early into my PhD, were also much appreciated. It would be remiss of me not to acknowledge the various proofreaders and anonymous reviewers who helped strengthen my publications, and the departmental administrative staff for all the logistical support over the years.

My friends at Warwick were an invaluable support system through it all. Lots of love goes out to Nicole Hengesbach, for all the conversation and Netflix breaks and vitamin reminders, and to Xin Zhi, whose shared enthusiasm for spicy food and my board games made even a pandemic lockdown enjoyable. Cheers to Teddy Cunningham, Matteo Mazzamurro, Chris Conlan, Nazem Khan, and everyone else from my department, house shares, and social groups, for all the technical discussions, levity, and fellowship.

Immense gratitude goes to Ma and Baba for their pragmatic advice and unwavering parental love, and to my cousin Julius for being a London home base. Finally, I thank my oldest and dearest friends across various time-zones who knew exactly when to check in on me, or who let me use our Dungeons and Dragons sessions as a creative outlet and break from research.

Declarations

I, Aparajita Haldar, declare that the work included in this thesis is my own, unless otherwise stated below. This thesis has not been submitted for a degree at any other university.

This work was funded by the Feuer International Scholarship in Artificial Intelligence.

Parts of the results presented in this thesis have been previously published in journal, conference, and workshop papers as the following:

[66] **Aparajita Haldar**, Shuang Wang, Gunduz Demirci, Joe Oakley, and Hakan Ferhatosmanoglu. Temporal cascade model for analyzing spread in evolving networks. *Transactions on Spatial Algorithms and Systems (TSAS)*, 2023

Contribution: The author of this thesis contributed towards designing the solutions and evaluations, and preparing the manuscript.

[39] Gunduz Demirci, **Aparajita Haldar**, and Hakan Ferhatosmanoglu. Scalable graph convolutional network training on distributed-memory systems. *Proceedings of the VLDB Endowment*, 16, 2022

Contribution: The author of this thesis contributed towards the solutions and preparing the manuscript, including conducting all the experimental evaluations on GPU clusters.

[184] Weiren Yu, Sima Iranmanesh, **Aparajita Haldar**, Maoyin Zhang, and Hakan Ferhatosmanoglu. Rolesim*: Scaling axiomatic role-based similarity ranking on large graphs. *World Wide Web*, 25(2):785–829, 2022

Contribution: The author of this thesis contributed the methodology, conducted the experiments, and provided the content for Section 7 “Scaling RoleSim* search using triangle inequality and partitioning” of the journal paper, and the corresponding evaluation results presented under “Unsupervised Semantic Evaluation (in Section 8.1) and “Semantic Accuracy” (in Section 8.2). Only these relevant findings are presented in this thesis.

[183] Weiren Yu, Sima Iranmanesh, **Aparajita Haldar**, Maoyin Zhang, and Hakan Ferhatosmanoglu. An axiomatic role similarity measure based on graph topology. In *Software Foundations for Data Interoperability and Large Scale Graph Data Analytics*, pages 33–48. Springer, 2020

Contribution: The author of this thesis contributed the content under "Semantic Evaluation" (in Section 4.1) and under "Semantic Accuracy" (in Section 4.2). These findings are a subset of the extended journal submission listed above.

Other parts of this thesis that are currently under submission are listed below:

[63] Rustam Guliyev, **Aparajita Haldar**, and Hakan Ferhatosmanoglu. D3-GNN: Dynamic Distributed Dataflow for Streaming Graph Neural Networks. 2022

Additional research that was performed during the development of this thesis but does not form part of the thesis are listed below:

[65] **Aparajita Haldar**, Teddy Cunningham, and Hakan Ferhatosmanoglu. RAGUEL: Recourse-Aware Group Unfairness Elimination. In *Proceedings of the 31st ACM International Conference on Information and Knowledge Management*, pages 666–675, 2022

[15] Vishwash Batra, **Aparajita Haldar**, Yulan He, Hakan Ferhatosmanoglu, George Vogiatzis, and Tanaya Guha. Variational recurrent sequence-to-sequence retrieval for stepwise illustration. In *European Conference on Information Retrieval*, pages 50–64. Springer, 2020

[167] Brendan Whitaker, Denis Newman-Griffis, **Aparajita Haldar**, Hakan Ferhatosmanoglu, and Eric Fosler-Lussier. Characterizing the impact of geometric properties of word embeddings on task performance. *NAACL HLT 2019*, page 8, 2019

Abstract

Graph-structured data has benefits of capturing inter-connectivity (topology) and heterogeneous knowledge (node/edge features) simultaneously. Hypergraphs may glean even more information reflecting complex non-pairwise relationships and additional metadata. Graph- and hypergraph-based partitioners can model workload or communication patterns of analytics and learning algorithms, enabling data-parallel scalability while preserving the solution quality. Hypergraph-based optimisations remain under-explored for graph neural networks (GNNs), which have complex access patterns compared to analytics workloads. Furthermore, special optimisations are needed when representing dynamic graph topologies and learning incrementally from streaming data.

This thesis explores hypergraph-based optimisations for several scalable graph analytics and learning tasks. First, a hypergraph sampling approach is presented that supports large-scale dynamic graphs when modelling information cascades. Next, hypergraph partitioning is applied to scale approximate similarity search, by caching the computed features of replicated vertices. Moving from analytics to learning tasks, a data-parallel GNN training algorithm is developed using hypergraph-based construction and partitioning. Its communication scheme allows scalable distributed full-batch GNN training on static graphs. Sparse adjacency patterns are captured to perform non-blocking asynchronous communications for considerable speedups (10x single machine state-of-the-art baseline) in limited memory and bandwidth environments. Distributing GNNs using the hypergraph approach, compared to the graph approach, halves the running time and achieves 15% lower message volume. A new stochastic hypergraph sampling strategy further improves communication efficiency in distributed mini-batch GNN training.

The final contribution is the design of streaming partitioners to handle dynamic data within a dataflow framework. This online partitioning pipeline allows complex graph or hypergraph streams to be processed asynchronously. It facilitates low latency distributed GNNs through replication and caching. Overall, the hypergraph-based optimisations in this thesis enable the development of scalable dynamic graph applications.

Abbreviations

1D	One-Dimensional
API	Application Programming Interface
BFS	Breadth-First Search
CPU	Central Processing Unit
CSR	Compressed Sparse Row
DGL	Deep Graph Library
DMM	Dense Matrix Multiplication
ESM	Expected Spread Maximisation
FM	Fiduccia-Mattheyses
GCN	Graph Convolutional Network
GNN	Graph Neural Network
GP	Graph Partitioning
GPU	Graphics Processing Unit
HGNN	Hypergraph Neural Network
HP	Hypergraph Partitioning
IC	Independent Cascade
KL	Kernighan-Lin
LT	Linear Threshold
ML	Machine Learning
MPGNN	Message Passing Graph Neural Network
MPI	Message Passing Interface
NCCL	NVIDIA Collective Communication Library
nDCG	Normalised Discounted Cumulative Gain
NLP	Natural Language Processing
NP	Non-deterministic Polynomial-time
POI	Point of Interest
RP	Random Partitioning
RQ	Research Question
RS	RoleSim
RS*	RoleSim*
RSM	Reverse Spread Maximisation
SHP	Stochastic Hypergraph Partitioning
SOTA	State-of-the-Art
SpMM	Sparse-dense Matrix Multiplication

SpMV	Sparse Matrix-Vector multiplication
SR	SimRank
SSRS*	Single-Source RoleSim*
SSRS*-PA	Single-Source RoleSim* with inter-partition access
SSRS*-P	Single-Source RoleSim* without inter-partition access
T-IC	Temporal Independent Cascade

Symbols

Symbol	Description
A	Active set of nodes
$\boldsymbol{A}(i,j)$	Entry in row i and column j of a matrix $\boldsymbol{A}$
$\boldsymbol{A}$	Adjacency matrix (normalised)
a	GNN aggregated feature
$\mathcal{a}(\cdot)$	GNN aggregator function
α	Learning rate of a model
b	Number of mini-batches
β	Decay factor for RoleSim
c	Damping factor for SimRank
$\chi(\mathit{\Pi})$	Connectivity cut size under $\mathit{\Pi}$
$cols(\boldsymbol{A})$	Non-zero columns of a matrix $\boldsymbol{A}$
$cost(u,v)$	Cost of an edge (u,v)
$cost(n)$	Cost of a net n
$deg(S)$	Total degree of a subset S (sum of node degrees)
$deg(v)$	Degree of a vertex v
δ	Significance (probability of error)
$\boldsymbol{D}$	Diagonal matrix of vertex degrees
d	Number of dimensions (or features)
$\mathcal{d}(u,v)$	RoleSim* distance metric for node-pair (u,v)
e	Edge (also called link)
$\mathcal{E}$	Edge set of a graph
$\mathcal{E}_C$	Set of cut edges
ϵ	Error margin of approximation
η	Sampling ratio for duplicate node
$\mathbb{E}$	Expected value
$\mathcal{f}(u,v,t)$	Force of infection
$\gamma(\cdot)$	GNN message function
$\boldsymbol{G}$	Gradient matrix
$\mathcal{G}$	Graph (also called network)
g	Graph realised from $\mathcal{G}$
$\boldsymbol{H}$	Feature matrix
$\mathcal{H}$	Hypergraph
$\boldsymbol{I}$	Identity matrix
I	Indicator random variable

$\boldsymbol{J}$	Loss function
$\nabla_{\boldsymbol{H}}\boldsymbol{J}$	Matrix of derivatives of $\boldsymbol{J}$ with respect to the features in $\boldsymbol{H}$
$\mathscr{k}$	Solution set size (may be for top-$\mathscr{k}$ results)
$\lambda(n)$	Connectivity of a net n
$\Lambda(n)$	Connectivity set of a net n
ℓ	GNN layer number
L	Total number of GNN layers
$\boldsymbol{L}$	Incidence matrix
$M(u, v)$	Maximum bipartite matching for node-pair (u, v)
$\boldsymbol{M}$	Local masking matrix
m	GNN message
μ	Relative weight balancing factor for RoleSim*
$N_{in}(v)$	Neighbourhood of a vertex v (incoming edges)
$N_{out}(v)$	Neighbourhood of a vertex v (outgoing edges)
n	Net (also called hyperedge)
$\mathcal{N}$	Net set of a hypergraph
$\emptyset$	Empty set
ω	Explosion factor
$\odot$	Element-wise multiplication (Hadamard product)
$\otimes$	Multiplication performed under a semiring defined by GraphBLAS
Π	Partitioning of a graph
p	Number of parts in a p-way partitioning
$\phi(\cdot)$	Reverse spread function
$pins(n)$	Set of vertices connected by n (also called pins of n)
$\mathscr{p}$	Probability
$P(t)$	Probability distribution at time interval t (dynamic)
$\mathscr{p}(u, v, t)$	Probability along edge (u, v) at time interval t (dynamic)
$\mathscr{p}(u, v)$	Probability along edge (u, v) (static)
P_m	Processor number m
$\psi(\cdot)$	GNN update function
$\boldsymbol{Q}$	Backward transition matrix
q	Query node
q'	Sampled clone of node q
$\mathbb{R}$	Set of real numbers
$\rho(\cdot)$	Non-linear activation function
$\rho'(\cdot)$	Derivative of the activation function ρ
$RS(u, v)$	RoleSim similarity score for node-pair (u, v)
$rows(\boldsymbol{A})$	Non-zero rows of a matrix $\boldsymbol{A}$
$RR(u, g)$	Random reachable set from node u in g
$R(u, g)$	Reachable set from node u in g
$\boldsymbol{S}_1$	SimRank similarity matrix
$\boldsymbol{S}_2$	RoleSim similarity matrix
$\sigma(\cdot)$	Spread function
$SR(u, v)$	SimRank similarity score node-pair (u, v)
$s(u, v)$	RoleSim* similarity score (exact) for node-pair (u, v)

$s_P(u, v)$	RoleSim* similarity score (approximate, without inter-partition access) for node-pair (u, v)
$s_{PA}(u, v)$	RoleSim* similarity score (approximate, with inter-partition access) for node-pair (u, v)
S	Subset of $\mathcal{V}$ (may be a solution set of nodes)
Σ	Summation operator
ϑ	Imbalance tolerance for partitioning
t	Time interval within the time window T
T	Time window
(u, v)	Node-pair (may be an edge)
v	Vertex (also called node)
$\mathcal{V}$	Vertex set of a graph
$\mathcal{V}_S$	Vertex separators (also called vertex cut or separating set)
$w(e)$	Weight of an edge e
$\boldsymbol{W}$	Weight matrix (also called trainable parameter matrix)
$\Delta\boldsymbol{W}$	Matrix of derivatives of $\boldsymbol{J}$ with respect to the parameters in $\boldsymbol{W}$
$W(\mathcal{V})$	Weight of a part $\mathcal{V}$
W_{avg}	Average weight of each part
$w(v)$	Weight of a vertex v (may denote computational load)
x	Feature (may be a node embedding)
ξ	Distance threshold for pruning RoleSim*

Chapter 1

Introduction

> *"Exactly!" said Deep Thought. "So once you do know what the* ***question*** *actually is, you'll know what the answer means."*
>
> Douglas Adams, *The Hitchhiker's Guide to the Galaxy*

Graph structured data is omnipresent today and has widespread applications – recommendation systems for e-commerce, classification algorithms for physical/biological systems like protein molecules, forecasting models for road traffic, and analytics algorithms for social networks to name a few. Entities and their relationships can form large-scale inter-connected networks where the topological structure captures information that cannot otherwise be represented. New graph data is being constantly generated from sources such as social media and wireless sensor devices. Given the ever-expanding sizes of graph data, it is becoming progressively more difficult to process an entire graph on a single machine. Therefore, it is prudent to explore techniques either to sample smaller subgraphs from the data for a single machine to handle, or to partition the data into parallel, distributed tasks that can be run on multiple machines.

Furthermore, while graphs already offer a compelling mix of connectivity information alongside raw feature information, there are even more enticing avenues now within reach: temporal graphs can encode evolving connectivity patterns, heterogeneous graphs can marshal together data from multi-modal sources, and meta structures such as meta-paths or hypergraphs can reveal more complex patterns hidden in the data like non-pairwise relationships [168]. Hypergraphs, in particular, with their ability to generalise edge connections and thereby encode relationships between any number of vertices as 'hyperedges', can be used to develop sophisticated models of graph connectivity and communication patterns. Sampling techniques to construct hypergraphs, and partitioning techniques based on hypergraph models, are therefore valuable tools to enable highly scalable applications. This thesis examines hypergraph-based optimisations that can be applied to a variety of analytics and learning tasks on large graphs. The goal is to develop support for efficient, scalable,

distributed applications on real-world large-scale graph structures, keeping in mind various system considerations (e.g., limited memory, low network bandwidth, low latency).

For any large-scale data, the most common approach to improve the efficiency of the system is to distribute the data and computations. Distributed processing of large graphs is made possible by graph partitioning techniques, where the original graph data is divided into subgraphs that can be processed in parallel across several machines. Hypergraph-based partitioning models have been successfully wielded on a few well-known graph analytics workloads, and have been found to provide a more accurate representation of communication volume; this is due to their powerful capability of using hyperedges that connect any number of nodes [70]. However, there remains a need to evaluate the performances of partitioners on non-traditional workloads and even graph-based learning tasks.

Graph analytics solutions have been a staple in graph theory for centuries. Graph-based learning applications are a much newer trend, now that machine learning (ML) methods have established a stronghold for automated decision-making and predictive tasks. The ability to use graph data in deep learning is a relatively recent advancement. Graph neural networks (GNNs) operate directly on graph structures to learn vector embeddings and perform ML tasks such as node classification and link prediction. GNNs have seen a meteoric rise in popularity and form the core of numerous industry applications today. For example, the e-commerce giant Alibaba uses graph embeddings for product recommendations on a massive scale [161]. Uber Eats and Pinterest also curate their recommendations using GNNs [79, 179]. Amongst other things, graph learning has been used for fake news detection through network spread analysis [121], for drug discovery by modelling biomolecular structures [52], and for traffic forecasting on spatio-temporal networks [82].

One of the main challenges of GNNs, which prevents their wide adoption in industrial applications, is the poor scalability to graphs that contain billions of records having high-dimensional features. For instance, social graphs used by Facebook contain up to a trillion edges [34], and e-commerce graphs used by Alibaba consist of billions of user/item nodes [161]. What makes learning tasks on graphs different from analytics tasks is the unique access patterns afforded by the neighbourhood aggregation steps over multiple layers of GNN computations. The inter-dependencies between nodes lead to complications in mini-batch generation and excessive network communication overheads. While sampling approaches have seen some success, sampling can lead to bias and loss of meaningful structural information. Hence, distributed whole-graph training is often necessary to maintain the utility of the model while improving its efficiency on massive datasets.

An understanding of the access patterns arising from GNN computations provides an opportunity to adjust the data partitioning and communication schemes in a manner designed to serve GNNs effectively. Due to the relative infancy of the field, studies are still limited on data-parallelisation techniques that are tailored specifically towards GNNs. Thus, it is useful to determine intelligent partitioning strategies for the data and computations involved during graph learning (e.g., the underlying graph adjacency data, input feature matrices, and trainable model parameters) that can minimise the communication overheads during training and inference on distributed systems, especially in environments having memory

and network limitations. Hypergraph construction is well-suited to model communication patterns accurately for such uses, especially when the graph connectivity data is sparse.

In real-world settings, the massive scale of the graph data is not the only challenge. Many real-world graphs are also inherently dynamic, containing evolving patterns of connections, or features that change over time. Traditional graph algorithms are not built for temporal workloads, and the scalability and quality of their results often break down when faced with such inputs. More widespread support is desirable for graph analytics and graph-based learning applications to operate on dynamic graph data at scale. Temporally-guided sampling and partitioning techniques on hypergraphs are therefore vital and interesting challenges.

In a similar vein, while numerous GNN architectures have been devised for static graphs, vast amounts of training data are being continuously generated by events that denote temporally evolving relationships between entities (such as social network interactions or user-item preferences [166, 179]). Graph streams refer to data arriving in the form of a sequence of edge connections or vertices (the latter with their complete adjacency information). Incremental updates and online inference tactics may be employed for graph-based learning systems to process dynamic graph data with sufficiently low latency and high throughput. With this development of GNNs for dynamic graph streams comes a need for streaming partitioning techniques to permit efficient and scalable computations.

Given a stream of vertices or edges, heuristics are commonly used to decide the partition into which to place the new arrivals [155, 156]. Such a one-pass, streaming graph partitioning algorithm may be applied just prior to a data-parallel dynamic GNN model. However, streaming partitioners have been primarily evaluated for analytics tasks rather than GNN workloads. Graph-based learning models provoke new challenges such as 'neighbourhood explosion', calling for strategies involving replication of vertices and re-scaling of partitions to be incorporated within the streaming partitioning scheme. Therefore, there is ample scope for research into partitioners designed to fit within a low latency stream processing environment alongside dynamic GNNs. The ability to distribute streaming data on-the-fly unleashes the power to process large (even unbounded) graph streams with high throughput and minimal communication overheads. This allows incremental learning models to be run even on machines that have limited memory and network bandwidth.

Additionally, the inclusion of more complex data can create future prospects of using heterogeneous features and hypergraph metadata to encode streaming workload patterns for more intricate load balancing and communication scheme improvements. Existing streaming pipelines are restricted to edge and vertex streams, and none can handle complex inputs such as hypergraphs. Support for hypergraphs is particularly useful as it extends the scope of streaming graph pipelines to also process partial/incomplete adjacency information in a vertex-centric manner. For example, when a hyperedge is ingested, subsequent hyperedges of a vertex may introduce new previously unseen adjacent vertices. This is not possible with vertex streams where each vertex must be provided with its complete immediate neighbourhood. The use of such partial adjacency information is unsupported by current state-of-the-art methods. Therefore, streaming partitioners for graph-based learning, on both graph and hypergraph stream workloads, are a compelling area of study.

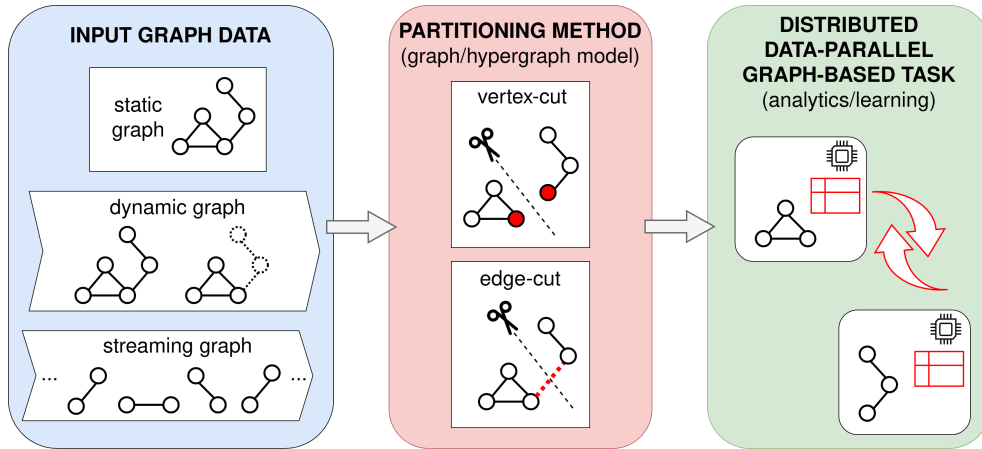


Figure 1.1: Illustrative pipeline where graph-structured data is ingested in static, dynamic, or streaming format, and distributed using an appropriate static or streaming partitioner (which uses a graph or hypergraph model), to allow distributed, data-parallel computations (static or online) for graph analytics or graph-based learning at large-scale

Overall, this thesis aims to develop more generalisable and efficient tools, for both graph analytics and learning, that can operate on real-world large-scale graph structures. Several hypergraph-based optimisations and transformations are explored to sample or partition graphs that may be static, contain temporal connectivity patterns, or be ingested as complex event streams.

1.1 Research challenges

This section provides an overview of the main research challenges addressed in the thesis. The three research questions (RQs) identify gaps in the literature with respect to the broader motivations laid out above, and outline the specific scope and objectives of the respective thesis chapters that tackle them.

Broadly, this body of work can be associated with three domains that have complementary connections with one another: graph/hypergraph partitioning, graph analytics/learning, and dynamic/streaming data. Each research question discussed below ties into one or more of these topics, and contributes towards the overall goal of this thesis. An illustrative pipeline of the combined work is presented in Figure 1.1.

RQ 1: Does the use of hypergraphs improve optimisations in sampling and partitioning for graph analytics and learning?

Hypergraphs generalise graphs by allowing hyperedges (nets) to connect any number of nodes. This helps to expose more complex relationships (beyond simple pair-wise connections) that cannot be represented in ordinary graphs. Constructing nets associated with a node offers an interesting technique to sample subgraph information. Hypergraph-based sampling approaches have been proposed within some static graph analytics algorithms such as information cascade (or social contagion) models [19, 37]. One focus of this research question is to identify a hypergraph construction approach that can be applied to model temporally evolving graphs. The randomness in the sampling stage during the hypergraph

construction can potentially be exploited to account for network dynamics, thereby reflecting temporal dependencies and changing connectivity patterns.

Hypergraph partitioning has also gained popularity due to its strength of modelling communication patterns accurately [70]. Hypergraph-based partitioners are widely used for balancing computational load while minimising communication cut size for graph analytics, yet there is a lack of research into their utility for GNN applications. This research question thus also seeks an evaluation of the efficacy of hypergraph partitioning compared to graph partitioning, first for different similarity search graph analytics workloads, and then for distributed full-batch GNN training on various real-world networks.

Mini-batch training in GNNs makes use of sampling as well, which typically generates random batches of nodes (i.e., subgraphs). It is interesting to consider whether hypergraph construction, in tandem with this random sampling to create subgraphs, can also better encode communication volumes to benefit distributed mini-batch training in GNNs.

RQ 2: How do the demands on hypergraph (and graph) partitioners differ when employed for scalable graph analytics versus scalable graph learning?

Graph algorithms have been a staple for various analytics purposes such as to discover community structures and shortest paths, to model diffusion through networks, and to measure similarities between entities. The partitioning quality of various graph- or hypergraph-based partitioners is typically evaluated by running PageRank [125] on the distributed graph. However, the workload and application setting have a large impact on the utility of a partitioning scheme. For example, PageRank iteratively updates the scores of every node in the graph. In contrast, a single-source similarity search algorithm only traverses the common neighbourhood required to reach the target node. This makes highly connected or central nodes more likely to be accessed over large query workloads. Hence, this research question highlights the importance of evaluating whether partitioning can be suitably applied for the specific workload under question. In situations where there is no communication of data across the partitioned subgraphs, it is also necessary to determine how the solution quality is affected. Furthermore, examining the utility of a hypergraph-based partitioning model over a graph-based model helps to assess whether the former is indeed a more promising approach to encode communication overheads accurately, be it for graph analytics or graph-based learning tasks.

Graph-based learning models, when compared to analytics algorithms, rely on entirely different access patterns when operating on the graph. During the feedforward and back-propagation phases in graph convolutional network (GCN) training, for instance, the convolution operation involves message passing and aggregation steps that induce irregular data accesses due to complex graph inter-connectivity. Existing systems use partitioning algorithms designed for traditional graph algorithm workloads (e.g., PageRank, connected components, or shortest paths), which do not take complex GCN data access patterns into consideration. Therefore, intelligent message passing strategies need to be employed to achieve a communication-efficient distributed data-parallel solution for GNN inference and training. Identifying and properly evaluating such a strategy is another important focus of this research question.

RQ 3: Can hypergraph (and graph) partitioning be applied effectively for distributed learning on dynamic/streaming graph data, particularly in low latency settings?

Many real-world graphs are inherently dynamic with evolving topology and features over time. In graph-based learning, support for dynamism in GNNs involves modifications to allow graph snapshots to be processed in sequence [126, 135, 164]. However, distributed GNN systems are mainly designed for static inputs by employing static graph partitioning, and suffer poor data locality since the graph data is loaded with every mini-batch iteration. Batch processing systems suffer poor latency when faced with streaming input, motivating the need for streaming graph systems instead to perform incremental updates and online inference in GNNs.

This research question underlines the importance of developing a streaming partitioner that can keep pace with a streaming GNN pipeline. The partitioner must be evaluated on GNN inference and training workloads rather than on the graph analytics tasks that are typically used to measure partitioner performance. This is because the workload of the task at hand informs the partitioning decisions, with ingested nodes and edges being placed in appropriate machines on-the-fly to have good locality and load balance. In particular, decisions surrounding replicated partitioning policies for vertices can affect the efficiency of distributed GNN computations. The irregular access patterns of GNN operations also incur additional communication overheads and suffer from 'neighbourhood explosion' during the message aggregation step when updating node representations [11]. Therefore, the partitioning scheme must be designed to combat this issue, and to successfully meet the throughput and latency needs highlighted above.

Another priority of this research question is to examine streaming hypergraph partitioning for a data stream that consists of hyperedges. It is of interest to study the duality between hypergraphs and bipartite graphs through various hypergraph transformations, to determine whether models indeed benefit from the more complex hypergraph structures used to represent incomplete evolving neighbourhoods.

The previous research questions interrogate partitioning and sampling techniques from the perspective of both graph analytics and learning tasks, with a special focus on hypergraph-based approaches. There was a preliminary consideration of dynamic graphs in RQ1, which is expanded here into a deeper investigation of streaming systems. Thus, this research question seeks to assess the design of streaming graph/hypergraph partitioners within a pipeline that performs graph learning on massive graph streams in a distributed setting.

1.2 Contributions

Several contributions are presented in this thesis that address various gaps in the literature and offer advancements to the field. This section summarises the contributions (C1–C6) and relates them to the various research challenges that were laid out previously.

C1 **Hypergraph-based sampling for graph analytics on dynamic graphs:** An efficient hypergraph-based sampling strategy is designed to capture temporal patterns of spread in a dynamic network with evolving connections. Such sampling and modelling

techniques have been previously used on static graphs, but it is nontrivial to apply them to dynamic settings while still preserving the solution quality guarantees. The proposed hypergraph sampling strategy takes into consideration temporal dependencies in the graph analytics task, thus expanding the scope of previous such models to dynamic graphs.

C2 **Application and evaluation of hypergraph-based (and graph-based) partitioning for graph analytics on static graphs:** A partitioning scheme with vertex replication is applied for scaling approximate single-source nearest neighbour search. The quality of the approximate results depends upon the amount of information contained within the local data of the subgraph partitions or communicated across. The impact of restricting inter-partition communication is thus studied, after such partitions are obtained from the graph or hypergraph partitioning methods, as an initial indication of the benefits of the hypergraph model in encoding communication overheads. It is insufficient to rely on performance results from traditional workloads such as PageRank to determine the effectiveness of these partitioners for new algorithms and tasks. Therefore, an unsupervised experimental setting is devised to quantify the effectiveness of the similarity measure with different partitioning parameter settings.

C3 **Hypergraph-based partitioning for distributed full-batch GNNs on static graphs:** Graph and hypergraph partitioning strategies have been commonly evaluated on graph analytics workloads, but GNN training and inference steps induce irregular access patterns and thus pose unique challenges. The message passing and aggregation processes involved during the convolution step in GNNs incur heavy communication overheads. To minimise this network traffic, a hypergraph-based partitioning scheme is proposed with non-blocking point-to-point communications. The merits of the hypergraph model are demonstrated over the standard graph model for partitioning, since the latter does not accurately encode the communication costs. These optimisations, previously unexplored for GNN training, are shown to be highly scalable to large number of CPUs.

C4 **Hypergraph-based sampling for distributed mini-batch GNNs on static graphs:** When mini-batch sampling is performed in distributed GNN systems, it impacts the communication volumes across machines. This results in potentially lower quality of partitions that are based solely on full graph connectivity. Thus, a novel stochastic hypergraph partitioning model is proposed to encode the expected communication volume. This hypergraph sampling technique successfully captures the randomness of communication operations in distributed mini-batch training of GNNs and achieves improvements over the earlier hypergraph model.

C5 **Graph-based partitioning for distributed incremental GNNs on streaming graphs:** A partitioning scheme is designed within a streaming dataflow framework to support the distributed execution of a data-parallel streaming GNN pipeline. Due to the latency-critical nature of the incremental GNN pipeline that ingests graph stream events continuously, there is a need to optimise the latency of the streaming partitioner. This partitioner defines the replicated vertices to be used for inter-partition communications during the distributed GNN message passing step. It is also built to

support variable parallelism of the system to combat the neighbourhood explosion problem in GNNs and to allow flexible re-scaling of partitions.

C6 **Hypergraph-based partitioning for distributed incremental GNNs on streaming hypergraphs:** Streaming graph data is typically handled either as edge streams or as vertex streams (where each vertex is ingested along with its complete neighbourhood adjacency information). The more realistic situation of incomplete neighbourhoods in vertex streams, where subsequent streaming updates may introduce new neighbours, is ignored. Since there is an observed duality between such a vertex stream with incomplete neighbourhoods and a stream of hypergraph nets, a novel streaming hypergraph dataflow framework is proposed. This approach extends streaming support to complex hypergraph structures as input. Hypergraph transformations are used to examine the benefits of directly processing hypergraphs instead of using their corresponding approximate graph representations.

1.3 Thesis structure

In this chapter, details were provided of the main research questions being addressed, along with a summary of the main contributions of this thesis. The remainder of this thesis is structured as described below. The inter-relations between chapters, contributions, and research questions are outlined in Figure 1.2.

Chapter 2 introduces the background knowledge necessary for various technical aspects of the thesis. Chapter 3 provides a thorough summary of related work to place this research in the context of the wider literature. Chapters 4–7 describe the various research findings of this thesis, with the first two focusing on graph analytics tasks while the latter two deal with graph-based learning using GNNs. Chapters 5 and 6 handle graph applications in the static domain. Chapters 4 and 7 study the dynamic setting, through i) temporal networks represented by graph snapshots in continuous time intervals, and ii) event data streams of graphs (edge streams) or hypergraphs (hyperedge streams), respectively.

Chapter 4 tackles RQ1, with a hypergraph-based construction approach proposed for the graph analytics task of spread modelling (Contribution C1). Hypergraphs are employed to sample reachable sets of nodes in a temporal network. This has applications in efficiently modelling information cascades within dynamic graphs where the topology patterns and characteristics of spread may change over time. A novel use of hypergraph sampling also helps to encode temporal dependencies, paving the way for further analysis of streaming graph settings related to RQ3.

In Chapter 5, hypergraph (and graph) models for partitioning are explored and evaluated on a graph analytics workload of similarity search (Contribution C2). Here, a bespoke evaluation strategy is devised to judge the quality of the approximate solution obtained using two different methods – one with strict data locality (no inter-partition access) and one without (inter-partition communication is permitted). Noting the benefits of a hypergraph model as per RQ1, and studying the above tradeoffs for RQ2, helps compare the access patterns across different workloads. This chapter also introduces the use of vertex replication for caching the computed values during operations on the graph, which is revisited in Chapter 7.

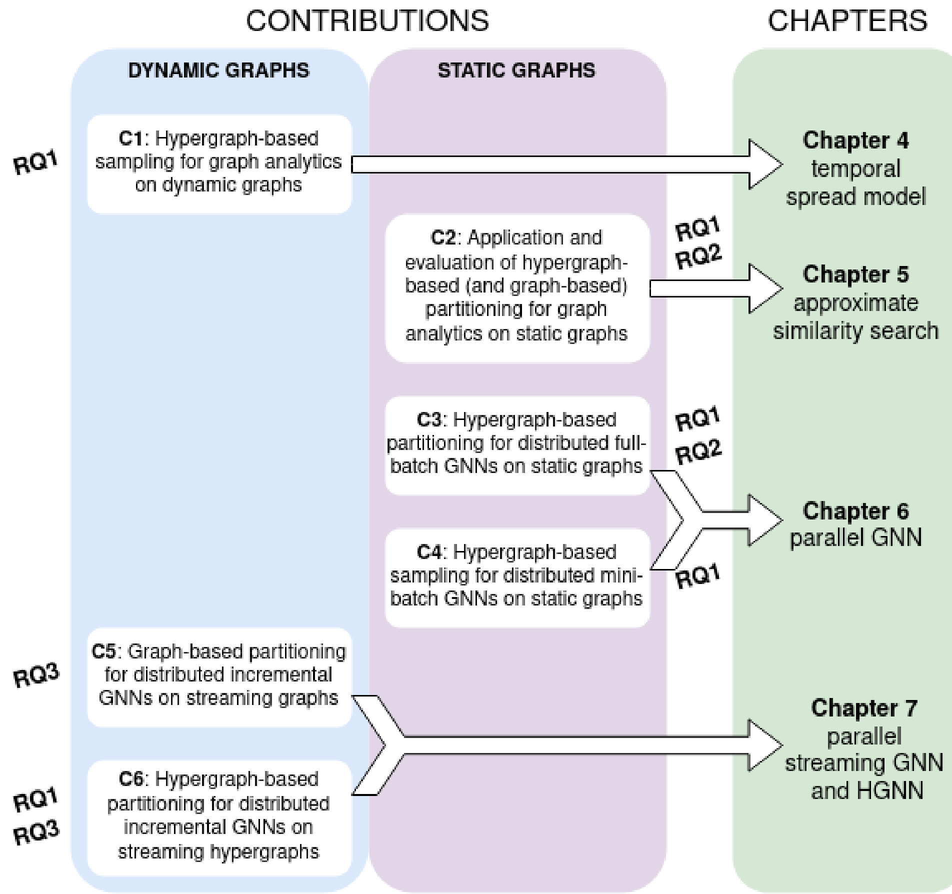


Figure 1.2: Thesis structure (chapters) and contributions with corresponding research questions (RQs)

Moving on to GNNs, hypergraph (and graph) models are used for partitioning in Chapter 6. The neighbourhood aggregation step in GNNs is found to induce complex access patterns, as per RQ2. To tackle this challenge, the work focuses on a novel communication-efficient algorithm using hypergraph models that can minimise network overheads during distributed full-batch training of GNNs (Contribution C3). The benefits of the hypergraph model over the graph model for encoding communication volumes are demonstrated, in an answer to RQ1. Furthermore, the hypergraph sampling ideas from Chapter 4 help to inspire a sampling-based approach to support distributed mini-batch training of GNNs (Contribution C4).

Chapter 7 confronts the low latency setting defined in RQ3. For a streaming dataflow framework with an incremental GNN model, the use of a streaming graph partitioner with replication provides distributed scalability, whilst meeting throughput and latency constraints of the system (Contribution C5). Moreover, the novel capability of ingesting complex hypergraph inputs, and using a streaming hypergraph partitioner for their distributed processing, offers functionality that is currently unsupported by state-of-the-art streaming systems (Contribution C6) and addresses RQ1.

Finally, Chapter 8 concludes this thesis with a summary of the contributions and key takeaways, followed by a discussion of limitations and opportunities for future work that arise from this research.

Chapter 2

Background

> *When action grows unprofitable, gather* ***information****; when information grows unprofitable, sleep.*
>
> Ursula K. Le Guin, *The Left Hand of Darkness*

This chapter provides the necessary background knowledge and preliminaries for the technical discussions in the remainder of the thesis. Further problem-specific preliminaries are included within subsequent chapters as needed.

2.1 Graph-structured data

Static graphs

Data can take various forms such as relational databases, images, text, or unstructured networks. In particular, graph data structures are non-Euclidean in nature, representing the data primarily in terms of the relationships between entities.

A graph $\mathcal{G}$ is denoted as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$ is a set of vertices and $\mathcal{E}$ is the set of edges $e = (v_i, v_j) \in \mathcal{E}$ representing pairwise relationships between these vertices.

In this thesis, the terms 'graph' and 'network' are used interchangeably, since the latter is commonly found in application domains. Similarly, 'vertices' are often referred to as 'nodes', and 'edges' may be called 'links'.

In a directed graph, edges have a source node and a target node, while in an undirected graph, they represent bidirectional connections between the node-pair. Nodes and edges may have any number of (heterogeneous) features. To simplify the presentation, a single feature can be considered associated with each node, denoted by $x_v \ \forall v \in \mathcal{V}$, and edge features can be similarly defined as $x_e \ \forall e \in \mathcal{E}$. If a graph $\mathcal{G}$ is weighted, edge weights can be treated as an edge feature where $w(e)$ is the weight of an edge $e \in \mathcal{E}$.

Any proper subgraph $\mathcal{G}_i$ is a graph whose vertices and edges are a subset of the original graph $\mathcal{G}$, that is, $\mathcal{G}_i = (\mathcal{V}_i, \mathcal{E}_i)$, where $\mathcal{V}_i \subseteq \mathcal{V}$, $\mathcal{E}_i \subseteq \mathcal{E}$.

Given a vertex v, its in-neighbourhood is defined as the set of all vertices in $\mathcal{V}$ which are connected by an incoming edge to v. Formally, the in-neighbourhood is $N_{in}(v) = \{u \in \mathcal{V} : (u, v) \in \mathcal{E}\}$. Out-neighbourhoods are similarly defined, but instead considering outgoing edges. For undirected graphs, there is no distinction made between in-neighbourhoods and out-neighbourhoods.

The adjacency matrix $\boldsymbol{A}$ of a graph $\mathcal{G}$ can be used to describe its topological connections, and is defined using entries $\boldsymbol{A}(i, j)$ such that

$$\boldsymbol{A}(i, j) = \begin{cases} 1, & \text{if } (v_i, v_j) \in \mathcal{E} \\ 0, & \text{otherwise} \end{cases}$$

Dynamic graphs

Dynamic graphs have an evolving topology where vertices and edges may be added or deleted over time. That is, they can be viewed as temporal or time-varying networks where the links carry information about when they are active. While usages vary across literature, in this thesis the terms 'dynamic graph' and 'temporal network' are used to denote the same type of data.

In order to make use of existing static graph algorithms with minimal modification, a dynamic graph is often defined as a series of static graph snapshots, i.e., $\mathcal{G} = (\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n)$, where $\mathcal{G}_t = (\mathcal{V}_t, \mathcal{E}_t)$ for $t \in T$ and $T = \{1, 2, \ldots, n\}$ is the time domain (i.e., set of all snapshots or time intervals).

For a more nuanced view of such temporal networks, it is possible to define either contact sequences or interval graphs. The former is useful where interaction durations are of negligible length and each contact (link) simply has a set of timestamps at which it is active. The latter instead uses a set of intervals to denote the start and end times, with links remaining active for the duration.

Graph streams

Data that is continuously generated, usually from different multi-modal sources, is referred to as streaming data or an event stream. Stream processing techniques allow for real-time, up-to-date analytics, training, and inference on such data.

A graph stream is data that is typically modelled as either edge streams or vertex streams. In edge streams, data is ingested as an asynchronous list of edges $(u, v) \in \mathcal{G}$, that may contain additional feature information. Vertex streams instead take the form of a list of tuples $(u, v_1, \ldots, v_k)$ where the entire adjacency information (local neighbourhood) of u is provided, i.e., $(u, v_1), \ldots, (u, v_k)$ are all the (directed or undirected) edges from u.

Streaming graph data has a wide variety, ranging from social network interactions and e-commerce purchase histories to spatio-temporal mobility data. Dynamic graph structures

may be generated by an asynchronous stream of timestamped events such as these. Each such timestamped event may be an add, delete, or update operation on a graph element (vertex, edge, or feature). In this manner, even unbounded graph streams can be processed, e.g., in limited memory systems or for low latency applications.

Hypergraphs

Hypergraphs generalise graphs by allowing hyperedges (nets) to connect any number of vertices. $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ can be used to denote a hypergraph consisting of vertex set $\mathcal{V}$ and net set $\mathcal{N}$. The set of vertices connected by a net $n_j \in \mathcal{N}$ is referred to as the pins(n_j).

Every hypergraph has an incidence matrix, whose rows correspond to the nodes and columns correspond to the nets. For an undirected hypergraph, the incidence matrix $\boldsymbol{L}$ is defined with entries $\boldsymbol{L}(i, j)$ such that

$$\boldsymbol{L}(i, j) = \begin{cases} 1, & \text{if } v_i \in n_j \\ 0, & \text{otherwise} \end{cases}$$

Hypergraphs have been found to be computationally more efficient requiring fewer matrix multiplication operations, and the incidence matrix of a hypergraph requires less storage space compared to the corresponding graph incidence matrix [168]. There is a duality between hypergraphs and incidence graphs. An incidence graph is a bipartite graph that can be constructed by taking all elements of $\mathcal{V}$ and $\mathcal{N}$ as nodes, and adding edges (v, n) if and only if $v \in n$, where $v \in \mathcal{V}$ and $n \in \mathcal{N}$.

There are two important strategies used to reduce a hypergraph to its dual graph: i) *clique expansion* which transforms a hyperedge into a clique of the multiple edges involved, and ii) *star expansion* which converts the hypergraph into a heterogeneous (bipartite) graph with nets as a second node type [200]. Star expansion is equivalent to constructing the incidence graph of the hypergraph. The main motivator for these transformations is the widespread availability of graph-based approaches. The expansions to convert hypergraphs into such graph structures allow existing graph analytics and learning algorithms to be directly applied.

However, it is important to note that hypergraphs are a nontrivial extension of graphs. This is because most, but not all, bipartite graphs can be regarded as incidence graphs of hypergraphs. It is only true if disallowing edges that are fully contained within other edges, but allowing repeated edges and edges containing one/no vertices [18]. In other words, graphs are a special case of hypergraphs. Therefore, reducing the hypergraph problem into a corresponding graph problem is inadequate as the resulting solution cannot fully capitalise on the information contained in the hypergraph. It has also been proven that an exact 'cut-model' representation of a hypergraph as a graph is impossible [78].

In Figure 2.1, some hypergraph-based sampling and transformation techniques relevant to this thesis are illustrated. Sampling subgraphs and denoting them as hyperedges, as shown in the top half of the figure, can be useful to generate hypergraph models from graph-structured data. For example, to model information cascades propagating through a network, given some initial seed (node v_3 in green), various reachable sets of nodes can be sampled as subgraphs of the original graph. These subgraphs, treated as nets, can then be merged to construct a

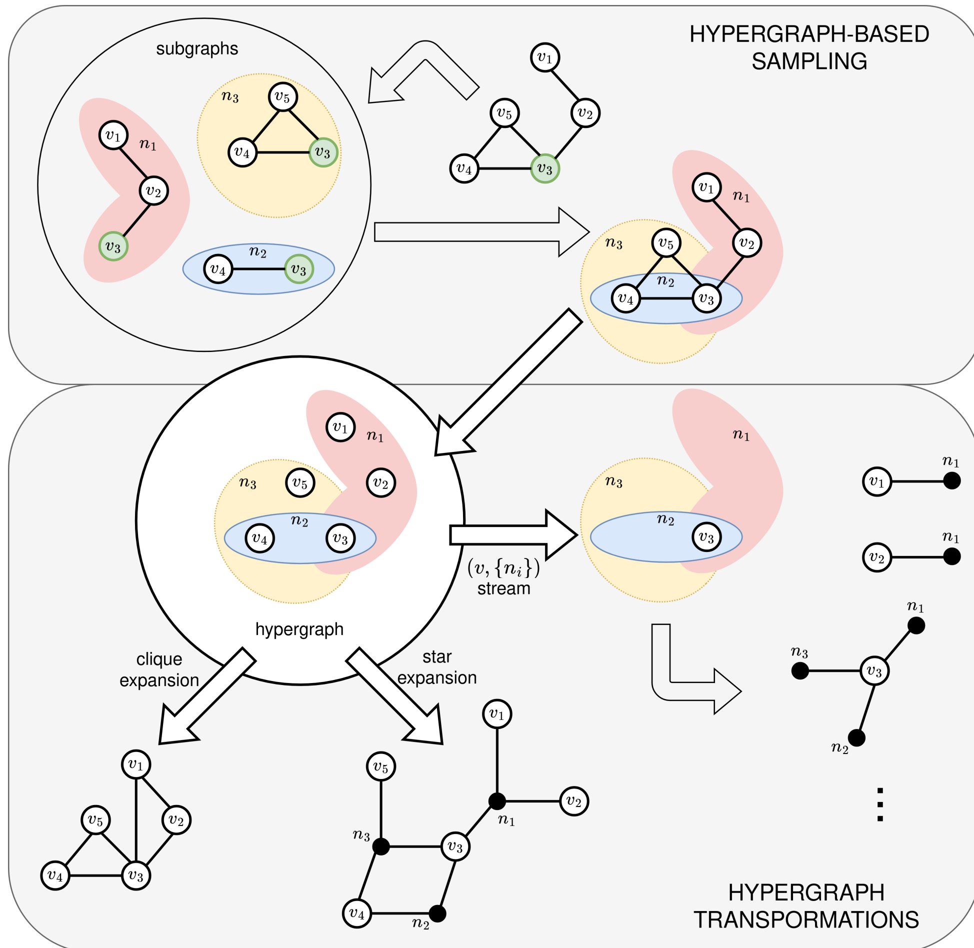


Figure 2.1: Hypergraph-based sampling and hypergraph transformation techniques

hypergraph that models the many possible paths of spread. Similarly, various mini-batch sampling technique for GNNs generate subgraphs by sampling the neighbourhoods of nodes; these subgraphs as nets can also form a larger hypergraph that models the message passing behaviour within the mini-batches during GNN inference or training.

Rather than constructing hypergraphs, hypergraph transformations to generate graphs or streams are also of interest. The resulting graphs from these approaches can be utilised as inputs for GNNs or any other systems that only operate on simple graph-structured data. A few hypergraph transformation techniques are shown in the bottom half of Figure 2.1. These transformations include the aforementioned clique expansion and star expansion. Note that these approximations fail to capture hierarchical higher-order relations. Clique expansion loses information about net n_2 because it is contained within n_3, while star expansion does not retain any vertex-vertex information and requires the algorithm to be designed to handle its heterogeneous nature. Another strategy is to convert the hyperedges into a stream of heterogeneous vertex adjacency information $(v, \{n_i\})$, which is especially helpful if the hypergraph input itself is ingested as a stream. For example, node v_3 belongs to nets n_1, n_2, and n_3, and is joined by edges to a new set of vertices that each denote a net. Each such vertex with its net list can be processed in sequence, in an approach that may be viewed as a dynamic version of star expansion.

2.2 Partitioning graph-structured data

To determine how a large-scale input dataset may be divided across processors for data-parallel computation, partitioning techniques are typically employed. Partitioning graphs is crucial since it allows for parallelisation or complexity reduction of a vast swathe of graph-theoretical problems and graph-based applications. It is fundamental for parallel and distributed frameworks to achieve scalable graph analytics or graph learning, given the increasing availability of large-scale graphs that cannot be loaded into memory in their entirety. Partitioning for graph-based applications is challenging due to complex inter-dependencies and access patterns between nodes.

Graph partitioning is the mathematical technique of dividing a graph into subgraphs while satisfying various constraints. Graph partitioning is generally performed by identifying 'cut edges' or 'cut vertices' that can be removed or replicated to create disconnected subgraphs, i.e., partitions or parts. Cut edges are the edges of the original graph that connect nodes from different partitions, and can provide an estimate of the amount of information exchange that may be necessary across partitions during distributed computations on the partitioned graph. Similarly, cut vertices offer an estimate of the amount of replication of nodes that may be required.

It is desirable for the computational load to be balanced across processors and for communication overheads across processors to be minimised. Put simply, the graph must be divided into equal sized parts while the number of cut edges or cut vertices that are left spanning different parts is minimised. That is, the two main constraints that are optimised when partitioning a graph are as follows:

i) *Balanced partitions*: Graph data, and the computational tasks associated with every node, should be split evenly across processors (physical parts) such that data locality and balanced parallel workloads are achieved. Achieving such load balancing ensures that processors can minimise the idle time of waiting for synchronisation or communication from other busy processors.

ii) *Fewer cut edges or cut vertices*: Communication/replication operations, e.g., inter-partition message exchanges (in edge-cut partitioning) or synchronisation of node representations across master/replicas (in vertex-cut partitioning), cause significant overheads and should be minimised. These overheads must not eclipse the computation time, because that prevents scalability to high processor counts.

More refined partitioning techniques may add constraints specific to the application scenario. For example, in a streaming partitioning case where the input graph changes structure dynamically, it may be necessary to periodically re-balance the partitions. Here, the initial partitioning step might also aim to minimise subsequent re-balancing costs (cost of migrating nodes across partitions).

Graph partitioning

Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with vertex set $\mathcal{V}$ and edge set $\mathcal{E}$, a p-way (edge-cut) partitioning of $\mathcal{G}$ is defined as $\Pi = \{\mathcal{V}_1, \mathcal{V}_2 \cdots \mathcal{V}_p\}$ consisting of subsets of vertices $\mathcal{V}_m \subset \mathcal{V}$ that are

mutually disjoint ($\mathcal{V}_m \cap \mathcal{V}_n = \emptyset$ if $m \neq n$) and nonempty ($\mathcal{V}_m \neq \emptyset \; \forall \mathcal{V}_m \in \Pi$) where the union of these subsets gives the vertex set ($\bigcup \mathcal{V}_m = \mathcal{V}$).

Each edge $(v_i, v_j) \in \mathcal{E}$ between vertices $v_i, v_j \in \mathcal{V}$ is given a cost(v_i, v_j) and each vertex $v_i \in \mathcal{V}$ is associated with a weight $w(v_i)$, therefore the weight of a part $\mathcal{V}_m \in \Pi$ is defined as $W(\mathcal{V}_m) = \sum_{v_i \in \mathcal{V}_m} w(v_i)$.

The partition Π is balanced if it satisfies

$$W(\mathcal{V}_m) \leq (1 + \vartheta) W_{avg}, \quad \forall V_m \in \Pi$$

where $W_{avg} = \sum_{v_i \in V} w(v_i)/p$ is the average part weight and ϑ is the maximum allowed imbalance ratio.

In edge-cut partitioning, vertex sets are disjoint and thus vertices are unique to single partitions whilst edges may cross partitions and therefore be shared between two partitions. Under such a partitioning Π, an edge $(v_i, v_j) \in \mathcal{E}$ is called a cut edge if it connects vertices belonging two different parts. The p-way graph partitioning problem is defined as finding a balanced partitioning Π such that the total partitioning cost is minimised. In other words, the partitioning must satisfy the above balancing constraint while minimising the global cost associated with the set of cut edges $\mathcal{E}_C$, i.e.,

$$\min_{\Pi} \sum_{(v_i, v_j) \in \mathcal{E}_C} \text{cost}(v_i, v_j)$$

Vertex-cut partitioning is similar, but instead defined as a disjoint family of edge sets $\mathcal{E}_m \subset \mathcal{E}$. Thus, in vertex-cut partitioning, vertices may be shared between two or more partitions, while the edges are restricted to single partitions. In other words, cut vertices may get replicated to reside in multiple partitions. Hence, 'replication factor' is often used to define the global cost associated with cut vertices in terms of the number of replicas created. More formally, replication factor is typically computed and minimised as

$$\min_{\Pi} \frac{\sum_{i=1}^{p} |\mathcal{V}_i|}{|\mathcal{V}|}$$

Hypergraph partitioning

Given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ with vertex set $\mathcal{V}$ and net set $\mathcal{N}$, its partitioning Π can be defined as above. Each net $n_j \in \mathcal{N}$ is associated with cost(n_j). The weight $W(\mathcal{V}_m)$ of a part is defined similar to the graph partitioning case, as is the associated balancing constraint.

Under the partition Π, the connectivity set $\Lambda(n_j)$ is the set of parts that net n_j connects (i.e., pins$(n_j) \cap \mathcal{V}_m \neq \emptyset$). The number of such parts is called connectivity $\lambda(n_j) = |\Lambda(n_j)|$. If a net n_j connects to multiple parts, i.e., $\lambda(n_j) > 1$, it is said to be cut. Otherwise, the net is said to be uncut. The connectivity cut size under Π is defined as

$$\mathcal{X}(\Pi) = \sum_{n_j \in \mathcal{N}} \text{cost}(n_j) \times (\lambda(n_j) - 1)$$

The hypergraph partitioning problem is therefore finding a p-way partition that satisfies the balancing constraint while minimising this connectivity cut size. By allowing a net to connect multiple vertices, it may span more than two partitions (since a net is considered to connect to a part if it connects to any vertex in that part). This is unlike traditional graphs where a cut edge only crosses two different partitions. However, a hypergraph regards multiple cut edges between the same two parts as a single cut net. This departure from traditional graph partitioning allows communication costs across partitions to be more accurately modelled in situations where message packets would not be duplicated or transmitted redundantly to the same partition.

Hypergraph-based partitioning models are particularly well-suited for parallel sparse matrix-vector multiplication (SpMV) and sparse-dense matrix multiplication (SpMM) [26, 71]. These operations are frequently employed within scientific computing, and GNN architectures in particular see rampant use of SpMM along with dense matrix multiplication (DMM).

2.3 Graph analytics tasks

Graph analytics is used to analyse the relationships between entities. The field involves network analysis tools such as centrality measures, community detection, and analysis of connectivity or paths. These techniques have been employed in a wide range of applications including recommendation systems, social network analysis, and disease tracking.

In particular, this thesis focuses on two tasks: i) similarity search, which is widely used for retrieval and recommendation, and ii) cascade models, which have applications in social networks and contact networks.

Similarity search

The similarity between nodes in a graph is based on a comparison of their neighbourhood structures, i.e., the topology of all immediately adjacent nodes. Some more complex definitions may take into account information beyond simply the graph layout. Similarity search has many applications ranging from determining user-item preferences for recommendation systems to retrieving similar documents and webpages.

Typically, a similarity measure can be computed between a pair of nodes as some value indicating the match between them. Some measures such as cosine similarity or Jaccard similarity are also commonly applied to any vector representations having high-dimensional features. Other methods determine similarity based on structural context, such as SimRank [80] which is based on the idea that ‘similar objects are referenced by similar objects’.

Cascade models

Research on modelling cascades in networks has traditionally been focused on identifying the influential nodes in social networks, particularly for viral marketing campaigns [42]. This task, called influence maximisation, is to find the small subset of seed nodes in a graph such that it would maximise the expected spread of influence.

Two main types of cascade models, namely the independent cascade (IC) model and the linear threshold (LT) model, have been considered to study the spread through a network and the growth of the final influenced set of nodes. In IC models, each node has one chance with some probability of influencing its neighbour. In LT models, a node becomes active if the collective influence from its neighbours exceeds a certain threshold.

In this thesis, the terms 'cascade', 'diffusion', 'propagation', 'flow', and 'spread' are all used to refer to the same concept of information moving through a network. This 'information' may be referred to as 'activation' or 'influence' where appropriate.

2.4 GNN models

One chief motivation for the development of graph-based learning models was to represent graph-structured data in low-dimensional vector space, i.e., through feature or representation learning that results in graph embeddings similar to the widely used word embeddings in NLP [61, 129]. Advancements in convolutional neural networks for computer vision led to further breakthroughs, finally resulting in graph neural networks. GNN models typically learn graph representations, i.e., node or edge embeddings, that are then used for a downstream task. This makes it possible to perform end-to-end ML tasks on graph data. In so doing, GNNs leverage both feature information as well as topological structure obtained from graph-structured data.

These powerful graph-based learning models have seen a rise in popularity due to the vast potential of graph data in various domains, such as social networks, knowledge graphs, and recommendation systems. Variants of GNNs such as graph convolutional networks, graph attention networks, and graph recurrent networks have been applied with success on numerous deep learning tasks at node-level (e.g., node classification, clustering), edge-level (e.g., link prediction), and graph-level (e.g., graph classification, matching).

Message Passing GNNs

The expressivity of GNNs for modelling inter-dependencies between nodes in a graph is achieved via message passing operations as a means to aggregate and update information. As per the Message Passing GNN (MPGNN) paradigm [56], the computation task for a GNN layer at each of its nodes can be viewed as message generation along each incoming edge with an aggregation operation at the receiving node, after which the node updates its representation for use by the next GNN layer. Below is a common MPGNN formulation where the GNN layer ℓ generates embeddings x_v for layer ℓ+1 at every node v:

$$
\begin{aligned}
m_e^{(\ell+1)} &= \gamma(x_u^{(\ell)}, x_v^{(\ell)}, x_e^{(\ell)}) \qquad \forall\ (u, v, e) \in N_{in}(v) \\
a_v^{(\ell+1)} &= \mathcal{A}(m_e^{(\ell+1)} : (u, v, e) \in N_{in}(v)) \\
x_v^{(\ell+1)} &= \psi(x_v^{(\ell)}, a_v^{(\ell+1)})
\end{aligned}
$$

The messages m_e are generated along each incoming edge in the node's in-neighbourhood ($e = (u, v) \in N_{in}(v)$), as some function of the features (x_u, x_v, x_e). All $|N_{in}(v)|$ messages

at v are then aggregated into a_v, which is used in combination with its old feature $x_v^{(\ell)}$ to generate the new representation $x_v^{(\ell+1)}$ for layer $\ell+1$.

Therefore, the behaviour of each GNN embedding layer can be expressed by defining the MESSAGE (γ), AGGREGATOR ($\mathcal{a}$), and UPDATE (ψ) functions. Typically, γ and ψ are neural networks in themselves, whereas $\mathcal{a}$ is one of *Concat*, *Sum*, *Mean*, *Min*, *Max*, *LSTM*, or *Attention* functions. A multi-layer GNN is constructed by stacking these computations. Finally, an output (neural network) layer after the last embedding layer can be used to generate decisions or predictions based on the task at hand (e.g., node classification, link prediction).

Graph convolutional networks

GCNs are the most commonly used type of GNN and have wide applications. Spatial models, that are based on the above message passing idea, have become more popular than the older spectral-based methods, that were inspired by signal processing techniques. This thesis focuses on two GNN models in particular.

The GCN model proposed by Kipf and Welling [89] is a generalisation of convolution operations applied to non-Euclidean data. During forward-propagation, the adjacency matrix and trainable weight matrix are used to compute a new feature matrix at every layer from that of the previous layer. A non-linear activation function is applied to obtain the features at every layer. Similarly, backpropagation uses the final loss function and weight matrices to compute the gradient matrices for each layer. These gradients are then used to update the weight matrices according to some model learning rate.

This computation can be viewed in a vertex-centric way as a message-passing operation from each of the neighbours of a node followed by an aggregation function, as described earlier by the MPGNN paradigm. That is, at every node v:

$$x_v^{(\ell+1)} = \rho\left(\sum_{N_{in}(v)} x_u^{(\ell)} \boldsymbol{W}^{(\ell)}\right)$$

where $x_v^{(\ell+1)}$ is the updated feature at v during layer $\ell+1$ computation, $x_u^{(\ell)}$ is the feature of its neighbour u, and $\boldsymbol{W}^{(\ell)}$ is the trainable weight matrix.

The GraphSAGE [67] algorithm instead performs neighbourhood sampling for aggregation, and thus behaves similar to GCN but using sub-sampled neighbours for training. It can be used as an inductive framework since inference is possible even on unseen nodes/graphs, whereas GCN needs the full graph structure and can be used for transductive applications. GCN uses the *Mean* aggregator and GraphSAGE uses a generalised aggregation function.

When full-graph training cannot be performed due to the large-scale nature of the data, the usual solution is to resort to mini-batch training. Mini-batching is carried out through neighbourhood sampling to construct each batch (where parameter weights are updated after each batch). GraphSAGE is a popular neighbourhood sampling method for such purposes. Computations are commonly supported across multiple processors that train on mini-batches in parallel (where parameter weights are synchronised before being updated for all processors). Distributed full-batch or mini-batch training is also achievable, by partitioning

the input graph and defining the required communication scheme. As is evident from the above, vertex-centric MPGNN models lend themselves well to graph partitioning techniques that can minimise communication overheads during the message aggregation phase through data locality of the node neighbourhood that needs to be accessed.

2.5 Graph processing systems

Parallel and distributed systems

Distributed systems consist of many autonomous processors, each with their own local memory, that communicate with each other by exchanging messages. The processors in a typical distributed system run computations in parallel to perform some shared task, making it important to achieve load balancing and minimisation of communication overheads.

Data-parallel graph processing generally employs partitioning to split the input graph while preserving data locality. Such parallelism presents unique challenges in GNN tasks due to the dependencies between input data samples (nodes). Model-parallelism is typically achieved by pipelining operators corresponding to each GNN layer, or pipelining the processing of individual samples or single neural computations within a layer. Hybrid-parallel models perform both data- and model-parallel computations on the input graph and GNN model.

Streaming dataflow systems

Batch processing systems suffer poor latency when faced with streaming input, thus streaming systems have been developed to handle such data. Popular streaming dataflow systems include Apache Spark [185] and Apache Flink [25]. Such streaming pipelines are designed to efficiently process bounded and unbounded data streams with low latency and high throughput, typically using distributed environments.

Chapter 3

Related Work

> “ *“I’ve been **thinking**, Hobbes”*
>
> *“On a weekend?”*
>
> *“Well, it wasn’t on purpose”* ”
>
> Bill Watterson, *Calvin and Hobbes*

This chapter offers a review and discussion of the related literature in order to establish the wider context for this research. The literature review helps to identify challenges and solutions that are closely related to the contributions of this thesis. More specific technical details of the related work are presented within the preliminaries in subsequent chapters where needed.

3.1 Graph algorithms

Graph analytics

Graph algorithms are in popular use for analytics tasks, based on graph-structured representations of pairwise connections between nodes. Numerous algorithms have been developed for efficiently finding shortest paths, centrality, node similarity, information cascades, flow dynamics, and community structures in large graphs. These lend themselves to graph analytics applications on a variety of data, from social networks to knowledge graphs.

One of the most popular graph algorithms is Google’s PageRank, which models web documents as nodes, and determines query relevance based on the links among these nodes [125]. Similarity search algorithms are useful for related problems of document retrieval and recommendation. The SimRank measure proposed by Jeh and Widom [80] led to a rise in popularity of graph-based similarity models. Many SimRank-like models have been developed for efficient computation of similarity between node-pairs – e.g., P-Rank [191], RoleSim [83],

MatchSim [105], CoSimRank [136] – that differ from one another in their treatment of neighbouring nodes and the specific heuristics used during their iterative computations. Therefore, each of these elicits different access patterns when traversing the neighbourhood to compute similarity scores. Not only are there different measures, but also different types of queries used for a given measure – e.g., single-source, single-pair, and all-pairs similarity – which impacts the access patterns and produces entirely different workloads.

There is also extensive work in the areas of information diffusion and influence maximisation, especially for social networks, due to the lucrative opportunities in viral marketing [31, 42, 62, 101]. Under the widely used IC and LT propagation models, finding a subset of users that maximises the expected spread is shown to be NP-Hard [86]. There are approximate solutions with near-optimal time complexity [150] and parallel algorithms to scale influence maximisation tasks for large-scale networks [109, 145]. Temporal networks are of particular relevance in the analysis of spreading processes since time-varying interaction edges and the relative ordering of interaction events can be highly relevant to the model. For example, both infectious disease spread and rumour propagation are affected by the dynamics and duration of contact between individuals [59, 91], and computer viruses stabilise or die out depending on dynamic interactions with removable devices [199]. While modelling spread, a few IC models do incorporate time constraints. For example, the continuously-activated and time-restricted (CT-IC) model is a generalisation where every active node can activate its neighbours repeatedly until a given deadline [88]. The latency aware (LAIC) model incorporates influencing delay probabilities within a time constraint for the spread [107]. However, such methods do not address evolving contact network topology or dynamically changing propagation properties along each connection.

Graph neural networks

Deep learning on graphs has gained significant attention in recent years. Zhang et al. [190] survey a variety of GNN architectures – graph recurrent neural networks, graph convolutional networks, graph autoencoders, graph reinforcement learning, and graph adversarial methods – dividing them into node-level and graph-level tasks, and note that modelling the evolving characteristics of dynamic graphs remains understudied. Similarly, Wu et al. [170] survey several different architectures and discuss applications ranging from traffic forecasting to recommendation systems. Other taxonomies split GNNs on the basis of the types of graphs handled [198]. All these reviews note that heterogeneous graphs, dynamic graphs, and scalability are particularly interesting challenges to tackle.

GCNs generalise the convolution operation, performed by convolutional neural networks on structured data (e.g., images, time-series), to graphs [21, 38, 118, 144]. GCNs are used in a wide range of data-intensive graph applications such as node classification [89, 117], traffic forecasting on road networks [180], and recommendation systems on user-item graphs [179].

Surveys that look at GCNs typically categorise them into spectral-based and spatial-based models, depending on the type of convolution operation performed [187]. Spectral model convolutions borrow from signal processing techniques to localise the graph signal (node features) in its spectral domain or vertex domain through linear combinations. Spatial model convolutions instead extend traditional Euclidean convolutions and generally perform some

aggregations of features in the node neighbourhood, making them computationally more efficient and localised, although spectral information is ignored. In particular, the GCN model proposed by Kipf and Welling [89] is a special variant that performs semi-supervised node classification on graphs, and clearly denotes vertex localisation as the aggregation of neighbourhood node representations. Therefore, this model is often considered to bridge the gap between spectral-based and spatial-based models.

Most GNN methods use such message passing and aggregation along edges as their fundamental operations, and can be treated as special cases of the previously described message-passing (MPGNN) framework [56]. Popular libraries for developing GNN models include Deep Graph Library (DGL) [162] and PyTorch-Geometric [50].

Hypergraph neural networks

Generalised hypergraph structures can make use of hyperedges to build more powerful representations, e.g., biological networks, chemical molecular structures [48, 94]. Hypergraphs possess the benefit of modelling complex, non-pairwise relationships. Due to the ability to model higher-order correlations within the data, hypergraph-based learning has been explored in computer vision tasks [77, 196]. Similarly, in social networks, high-order social relations such as triangle motifs between user-item nodes and session-based temporally connected user-item interactions have been used to build multi-channel convolutional networks for recommendation [171, 181].

Hypergraphs are also useful to encode multi-modal heterogeneous data. For example, deep learning on hypergraphs has been proposed that uses multi-modal features for object recognition [49]. This hypergraph neural network approximates the hyperedges through clique expansion; it is a spectral method that generalises convolution operations using a hypergraph Laplacian. The traditional GCN can therefore be regarded as a special case. Another method, HyperGCN, also uses clique expansion to turn the task into a graph learning problem, but with a more efficient hypergraph Laplacian where each hyperedge is linearly approximated by exactly one pairwise edge [173]. Agarwal et al. [3] compare different hypergraph approaches in terms of their reductions to graph construction and their associated Laplacian matrix for spectral computations, finding many to be equivalent.

However, Arya et al. [7] argue that hypergraphs are not a special case of graphs, therefore information can be lost during this reduction to a graph problem. They propose HyperSAGE, an inductive approach inspired by the popular GraphSAGE method, for deep learning directly on hypergraph structures. HyperSAGE uses a two-level message passing and aggregation technique, treating the neighbourhood of a node in terms of within the net (intra-edge) and also across nets (inter-edge), to fully utilise higher-order relations in hypergraphs.

3.2 Distributed systems

Distributed graph processing systems

Distributed systems have been widely employed for graph analytics. Pregel [113] and PowerGraph [57] are particularly popular for distributed graph algorithms. GraphX [172] moves

beyond such graph-parallel systems and exploits a data-parallel framework to efficiently express graph computations. Several graph analytics APIs, including GraphX, are built atop Apache Spark or similar frameworks, which face CPU utilisation bottlenecks. These vertex-centric models also suffer high communication overheads relative to computation and thereby have poor scalability to large number of machines [69, 111]. In contrast, graph- and block-centric models utilise local graph partition structure to reduce communication and scheduling [153, 174]. Exploiting sparse connectivity patterns can also achieve better data locality in data-parallel vertex-centric solutions, enabling efficient communication across thousands of processors.

Graph partitioning is widely employed for improving the efficiency of different types of queries [46, 138, 192], handling skewed workloads [177], reducing communication overheads [55], and achieving scalability in network bound applications [34]. Communication overheads depend upon both message volume and communication patterns among partitions [55]. This motivates the need for intelligent partitioning and communication schemes tailored to particular workloads. Some methods do take into account task-specific considerations during the partitioning stage, such as by determining partitioning strategies adaptively at runtime [45] or using an application-driven partitioning strategy [46]. However, there remains a lack of research into GNN workloads for optimisations in partitioning.

Distributed GNN systems

Graph learning tasks perform both forward and backward propagation of model parameters, involving ℓ-hop neighbourhood aggregations, which require different considerations in partitioning compared to that of traditional graph processing. To efficiently train GCNs, methods have been devised to restrict the neighbourhood considered by sampling. FastGCN [30] samples a specific number of vertices for aggregation in each layer independently to reduce the computational footprint in deeper GCNs, while a different adaptive method performs sampling conditional on previous layers [76]. Similarly, a vertex neighbourhood sampling approach is utilised in GraphSAGE [67]. Clustering algorithms like ClusterGCN [33] also restrict convolution operations and neighbourhood search to sampled subgraphs. Pruning and caching methods have been devised as well, to achieve accelerated GNN inference [116, 197]. All these efforts focus on reducing computation costs and memory requirements without sacrificing the quality of vertex representations.

Memory management and distributed training are essential for scalable deep neural networks [14, 40, 163, 188]. Various frameworks use distributed memory systems for parallel GNN training. On GPUs, NeuGraph [112] uses dataflow scheduling while Roc [81] employs dynamic regression-based partitioning to optimise communication, and G3 [108] leverages graph native operations for parallel graph processing. The DGCL [24] communication library instead reroutes communications to use fast links with vertex replication during distributed GNN training.

The Deep Graph Library (DGL) Python package is especially popular for GNN development using existing deep learning frameworks such as PyTorch and TensorFlow. It provides message passing functionality, batch and sparse matrix optimisations, and distributed CPU/GPU training capabilities. DistGNN [115] optimises DGL for full-batch training using a shared

memory implementation on CPU clusters, and employs a minimum vertex-cut graph partitioning strategy and delayed aggregate functions to reduce communications. As an alternative to whole-graph training systems, sampling approaches have been deemed useful to overcome the coordination and communication overheads through mini-batches [142]. In DistDGL [194], reduction of network communication traffic is achieved by partitioning and co-locating the vertex/edge features with their corresponding local partition data for distributed CPU mini-batch training. DistDGLv2 [195] achieves a hybrid design by placing data in distributed CPU memory and performing mini-batch training on GPUs, with asynchronous sampling to overlap CPU and GPU computations.

Various other solutions optimise the sampling step in different ways. AliGraph [176] uses distributed graph storage and optimised sampling operators to enable large-scale training. AGL [186] is a fault-tolerant system running on top of MapReduce for scalable GNN training. PaGraph [106] is built on top of DGL and introduces caching to accelerate sampling-based data-parallel GNN training. However, studies are limited on optimising GNN communications without sampling, especially when distributed whole-graph training is necessary.

Zheng et al. [193] argue that distributed mini-batch sampling in GNN systems suffer performance bottlenecks due to the sampling step; they thus propose ByteGNN to improve resource utilisation on CPUs with a partitioning strategy tailored for GNN sampling. The resource under-utilisation problem is exacerbated in GPUs since mini-batch sampling overshadows training time even further [142]. The communication architectures on CPU also involve different optimisation design decisions compared to GPU-based systems [119]. Hence, it is important to consider whether the solution is targeting CPU or GPU clusters.

Data-parallelisation for GNNs

The potential of exploiting data access patterns in GNN training has been recently highlighted as an open research question [96]. Sparse-dense matrix multiplication (SpMM) and dense matrix multiplication (DMM) are core kernel operations in GCN training. SpMM achieves convolution whereas DMM corresponds to propagating vertex-feature vectors through a single layer neural network. Therefore, considering the access patterns in these computations is important in improving the resource efficiency and scalability of GCN training. There have been improved solutions for parallel SpMM [93, 98, 140] and DMM [13] problems. However, the special requirements of combining SpMM and DMM for scalable GCN training and ensuring efficient forward propagation and backpropagation phases during training remain under-explored. In particular, communication operations incur high latency and bandwidth costs to aggregate feature matrices during forward propagation as well as to aggregate gradients and update parameter matrices during backpropagation.

While recent parallel/distributed algorithms achieve GCN training for GPU clusters and cloud systems [154, 194], these typically perform broadcast and allreduce types of collective communication operations. Sancus [127] is a recent model that adaptively avoids broadcast communications to reduce network traffic in data-parallel GNNs. However, parallel SpMM still requires broadcast-type communication, which is the main performance bottleneck in GCN training, due to its high memory and bandwidth costs. CAGNET [154] uses the aggregate memory of GPU clusters and the NCCL multi-GPU communication library to

asymptotically reduce communication costs of SpMM in full-batch GNN training by dividing dimensions of the iteration space across the training pipeline. It performs broadcasts among processors turn-wise to transfer vertex-features in small portions but incurs significant latency overheads. Sparse data communication and compression methods have been considered to alleviate the scalability issues of allreduce for larger models and processor counts [40, 47, 95, 104]. Despite improvements, such redundant data and message transfer causes unnecessary communication overheads. Therefore, a viable alternative is to design a mechanism that can utilise non-blocking point-to-point communications that move only the necessary data among processors.

Moreover, in GCN training, model parameter matrices are significantly smaller than the adjacency and vertex-feature matrices, so performance improvements in allreduce communication are not significant in the overall parallel execution time. Instead, efficient parallelisation of SpMM performed on the large graph data can lead to higher performance increase. The GE-SpMM algorithm [75] for GPUs allows integration with DGL for faster computation of GNNs by processing columns in parallel and directly operating on sparse matrix data in CSR format. Feat-Graph [74] co-optimises graph traversal and feature dimension computation to offer efficient CPU/GPU implementations of sampled dense-dense matrix product (SDDMM) and SpMM in GNN training, and reveals sensitivity to partitioner parameters. FusedMM [132] develops a general-purpose matrix multiplication kernel for graph embedding and GNN operations. FusedMM unifies the matrix multiplications into a single operation since SpMM is frequently directly followed by DMM, but the approach is only applicable to shared-memory systems. Besides such speedups to SpMM, another promising strategy can be to leverage the partitioning scheme for communication-efficient GNN training through asynchronous local partial SpMM computations.

3.3 Partitioning methods

Static graph partitioners

Partitioning algorithms are widely employed on graphs to enable data-parallelism for load balance and low communication. Graph partitioning can by achieved either by vertex-cut methods (i.e., assigning an edge to a partition with vertices replicated across partitions where necessary) or edge-cut methods (i.e., assigning a vertex to a partition with edges crossing partition boundaries where necessary). Graph partitioning is NP-Hard, therefore solutions to these problems generally involve heuristics and approximation algorithms (e.g., local search, spectral methods), many of which are summarised in an old survey by Elsner [43].

One of the earliest algorithms, the Kernighan-Lin (KL) algorithm, was developed for optimising circuit design [87]. KL-based heuristics have been widely used for graph partitioning, offering fast good quality results via iterative vertex swapping to find the locally minimum partition from an initial bipartition. Spectral partitioning and clustering approaches are a common alternative to such local search techniques. In spectral methods, a partition is derived from approximate eigenvectors of the adjacency matrix or using the eigen-decomposition of the graph Laplacian matrix, thus operating on a global view of the graph [64].

Multi-level algorithms have gained popularity due to quickly producing high-quality partitions [22, 85]. These algorithms work in multiple stages. Each stage collapses vertices and edges of the graph, partitions the smaller coarsened graph, and then projects the partitioning back and refines it for the original graph. The partitioning stage may be performed by any relatively simple methods, such as the KL algorithm above.

METIS [84] is a widely used graph partitioning tool that is based on this multi-level graph partitioning paradigm. It is available as standalone command-line programs or API routines that can be invoked within user-written programs in C/C++. The graph adjacency information is input in CSR sparse matrix format. The output is the part vector providing the partition number allocated to each vertex. The `METIS_PartGraphKway` function is used to partition the graph using multi-level p-way partitioning. The `ComputeVertexSeparator` function instead bisects the graph by computing the vertex separator, with these separator (cut) vertices being identifiable in the resulting part vector.

Static hypergraph partitioners

Hypergraph partitioning has also been explored alongside graph partitioning. A faster implementation of the KL algorithm has been introduced for hypergraph partitioning, called the Fiduccia-Mattheyses (FM) algorithm [51]. Multi-level approaches for hypergraph partitioning are more recent. They use clustering to coarsen the hypergraph, FM-based partitioning on the resulting supernodes, and projection back. The hypergraph partitioning problem for finding a p-way partition that satisfies computational load balancing constraints while minimising the communication cut-size is also NP-Hard. Tools such as PaToH [27] and KaHyPar [139] use heuristics to produce good quality solutions.

PaToH provides a stand-alone program or a library interface that utilises the multi-level hypergraph bisection algorithm for p-way partitioning. It uses its own plain text input format, listing the pins of each net of the hypergraph. The output of the `PaToH_Part` function is a part vector giving the allocated partition for every vertex. As an extension, rpPaToH [141] proposes a replicated partitioning tool for undirected hypergraphs, where a replicated FM heuristic is used within the recursive bipartitioning framework of PaToH. To achieve partitioning and replication simultaneously, an additional replication ratio constraint is used. The output is the list of part numbers for each vertex indicating all the parts that it is replicated to.

KaHyPar has similar functionality to PaToH, producing slightly better cut sizes but with a slower running time. However, Mt-KaHyPar [58] offers a multi-threaded parallel partitioning algorithm that is capable of processing extremely large hypergraphs with comparable solution quality, making it a useful choice for large-scale data.

Streaming graph partitioners

To handle dynamic graph updates, some methods modify existing static graph partitioning schemes and introduce re-balancing strategies. One approach aims to lower the repartitioning cost, i.e., the cost of migrating vertices across partitions so as to re-balance the

partitioning [123]. Other adaptive methods similarly try to prevent introducing new cut edges while minimising the repartitioning cost [2, 8, 44, 156].

Graph partitioning heuristics may incur heavy costs that affect the latency of the subsequent graph analytics tasks, therefore there has been work exploring how to partition the graph in a lightweight streaming fashion as it gets loaded to memory. This low latency, online approach is well-suited to dynamic graphs where offline repartitioning of the entire new graph snapshot is inefficient.

Processing streaming data is also of increasing interest for big data analytics where the arriving input must be ingested and processed on-the-fly. If the entire data cannot fit into the memory, it must be partitioned across distributed systems. Both the batched graph computational models (e.g., Pregel [113], Giraph [34]), and stream processing systems (e.g., Apache Flink [25]), can make use of streaming graph partitioning. Streaming graph partitioning approaches can be applied on graph data in the form of a vertex stream (a sequence of nodes along with their corresponding adjacency lists describing their neighbourhood) or an edge stream (a sequence of individual edges that are interconnected).

Stanton and Kliot [146] compare different vertex stream orderings (random, breadth-first, depth-first) used to first linearise the graph, and identify a weighted linear deterministic greedy (LDG) approach (assigning a node to a partition with the most neighbours, penalising crowded partitions) for generating balanced edge-cut partitions. FENNEL [155] introduces a unifying edge-cut framework that subsumes two popular heuristics for streaming balanced graph partitioning: assigning a node to a partition having i) the fewest non-neighbours, or ii) the most neighbours. It thus accommodates performance requirements that depend upon system/application characteristics, e.g., balanced communication traffic is likely to be more important in large clusters, whereas balanced computational load may be more relevant for small clusters.

HDRF [130] instead performs vertex-cut partitioning, where high-degree vertices are replicated first, thus exploiting skewed degree distributions. The goal is to cut high-degree vertices to create strongly connected components having low-degree vertices, and place these into their own partitions.

Many such streaming partitioning and repartitioning objectives are compared in a survey of partitioning strategies by Buluç et al. [23]. The partitioning quality of these various graph partitioning approaches is typically evaluated by running PageRank on the distributed graphs. However, the workload and application setting have a large impact on the utility of a partitioning scheme. Abbas et al. [1] perform an experimental comparison of streaming graph partitioning methods across different applications and datasets, using a unified distributed system on Apache Flink. They conclude that model-dependent techniques (e.g., FENNEL for vertex stream, HDRF for edge stream) offer better communication performance while data-model-agnostic methods (e.g., hash) trade off data locality for better balanced workloads. Pacaci and Özsu [124] draw important distinctions between vertex-cut and edge-cut partitioning methods based on streaming vertex or streaming edge systems, and evaluate different workloads for each setting. Their experimental study demonstrates that the topology of the graph and characteristics of the task-specific workload should be used to

inform the choice of partitioner. Despite these studies on various graph analytics algorithms (e.g., PageRank, connected components, shortest path), there is no work that evaluates the utility of streaming partitioners adapted for distributed graph learning.

Streaming hypergraph partitioners

The streaming setting has not yet been explored very widely in the context of hypergraph partitioning. The main challenge is that in a hypergraph, a vertex appears with its nets, which does not implicitly provide information about neighbouring vertices (unlike in graph streams where the connected vertex becomes implicitly known). Hence, the connectivity among the nets and the parts needs to be accounted for during the part assignment process.

The objective for streaming min-max hypergraph partitioning is to divide a set of items (vertices), having sets of topics (nets) associated with them, into partitions as they arrive sequentially. This is related to the min-max multi-way graph cut problem where the goal is to partition vertices such that the maximum of the number of cut edges of each part (i.e., maximum capacity rather than average or sum) is minimised [147]. A greedy item placement strategy is employed such that the maximum number of distinct topics (pins) covered by each part is minimised [4]. Another memory-efficient variant makes use of optimisations on the part-to-net and net-to-part information storage for additional speedups [151]. The min-max hypergraph partitioning problem is NP-Hard and the streaming setting provides even less information to the algorithm during part assignments. Nonetheless, the above algorithm appears to be the only streaming hypergraph partitioners currently in use [114].

Chapter 4

Hypergraph-based Sampling on Dynamic Graphs for Analytics

> *The only thing binding individuals together is ideas. Ideas mutate and* ***spread****; they change their hosts as much as their hosts change them.*
>
> Bernard Beckett, *Genesis*

This chapter provides the first preliminary exploration in this thesis of hypergraph-based optimisation approaches. Here, the focus is on eliciting the dynamic behaviour of a network by using a temporally-guided sampling strategy to construct the hypergraph. Several interesting graph analytics workloads involve solutions that depend upon the temporal properties of the network, such as evolving interactions within a social network. While there has been a great deal of research on analytics tasks for static graphs, dynamic graphs present new and unique challenges arising from temporal dependencies for which solutions are not yet well-studied. One such task is influence maximisation, and more broadly, the study of spread within networks.

Modelling the spread of influence, disease, or other information through a network depends upon its connectivity patterns and on the random seeds from where the information cascade process begins. Traditional approaches for modelling propagation in networks (e.g., of diseases, computer viruses, or rumours) cannot adequately capture temporal properties such as order/duration of evolving connections or dynamically changing likelihoods of propagation along connections. Temporal models on evolving networks are therefore crucial in many applications that need to analyse dynamic spread.

In this chapter, a temporal cascade model is designed with the help of a hypergraph-sampling technique. This hypergraph is used to encode the set of reachable nodes from the random seeds during an information propagation cascade process. Temporal dependencies in the propagation step, arising from the evolving topology of connections, are taken into account

during this hypergraph construction. This approach ensures that solution quality guarantees continue to be preserved even in the dynamic setting, unlike in most other time-evolving cascade models. In future chapters, the idea of using hypergraphs is extended to partitioning techniques as well as new sampling ideas for GNN training, first in static graphs, and finally in streaming scenarios that build further on the dynamic setup discussed here.

More concretely, this chapter presents the *Temporal Independent Cascade (T-IC)* model with a spread function that efficiently utilises a hypergraph-based sampling strategy. The temporally-guided sampling policy and the use of dynamic propagation rates along edges ensure that the algorithm realistically models the spread of activations. The spread function is proven to be submodular, with guarantees of approximation quality when simulating spread through evolving networks. This enables scalable analysis on highly granular temporal networks where other models struggle, such as when the propagation rate across connections exhibits arbitrary temporally evolving patterns. For example, a disease-spreading virus has varying transmissibility based on interactions between individuals occurring with different frequency, proximity, and venue population density. Similarly, propagation of information having a limited active period, such as rumours, depends on the temporal dynamics of social interactions in the network.

Algorithms on evolving networks typically focus on maximising influence [53, 68, 122, 149, 178]. Influence maximisation aims to identify the subset of seed nodes that can result in the maximum spread through the network. To the best of the author's knowledge, there is no prior work on capturing *sentinel* and *susceptible* sets in temporal cascade networks as studied in this chapter. The former aims for maximum coverage of the network via a small group of sentinel (detector) nodes, and the latter identifies the nodes most likely to receive information rather than spread it. The selection of sentinel nodes has been an important task in many applications on static networks where early detection of activation is beneficial, such as monitoring disease outbreaks [12], signalling wireless sensor network failures [41], and detecting malicious data transmissions [148]. Identification of a minimal solution set of sentinel nodes is of interest as resources are often constrained (e.g., expensive medical tests or wireless sensors). Susceptible nodes are also studied in the context of static networks. For example, failure points in communication networks may be triggered by the cascading failure and redistribution of data packets, and have been modelled stochastically for mitigation purposes [133]. Other studies on fake news detection find that recognising easily influenced users is more important to control the spread, rather than identifying the influential users [143]. It is important to note, therefore, that these objectives are distinct from the widely studied influence maximisation objective.

The dynamics of the spread over an evolving network topology introduces several challenges in the selection of both the above types of solution sets. The reachable sets determined using the proposed hypergraph-based sampling are shown to be effective in identifying sentinel nodes and susceptible nodes, making it possible to provide solutions for the several above application domains even with temporal networks. T-IC is shown to significantly outperform the alternatives in modelling both *if* and *how* spread takes place within a dynamic graph, and is efficient even on large graphs.

4.1 Preliminaries

The traditional IC model description is provided below, as well as the definition of a temporal network that is used in this chapter.

Independent Cascade model

In a standard IC model, information flows through the network via a series of cascades. Nodes may either be active (already influenced by the information that is propagating through the network) or inactive (either unaware of the information that is propagating but has not reached it by this point, or not influenced by the propagation that did reach it). The standard IC model assumes a static probability distribution over a static graph structure. This IC process is simulated over a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \wp)$ where each edge $(u, v) \in \mathcal{E}$ is associated with a constant probability function $\wp : \mathcal{E} \mapsto [0, 1]$, reflecting the likelihood of activation when nodes $u \in \mathcal{V}$ and $v \in \mathcal{V}$ have a common edge (e.g., a common meeting point in the location histories of two individuals). Propagation starts from an initial seed set in $\mathcal{V}$ (the only nodes active at step 0). Propagation takes place in discrete steps with each active node u during step i being given a single chance to activate its currently inactive neighbour v with some probability $\wp(u, v)$. That is, at every step $i \geq 1$, any node made active in step $i-1$ has a single chance to activate any one of its inactive neighbours. The process continues with nodes remaining active once activated, until no further propagation is possible. Therefore, this is a stochastic process that requires a large number of simulations to accurately determine the spread of information.

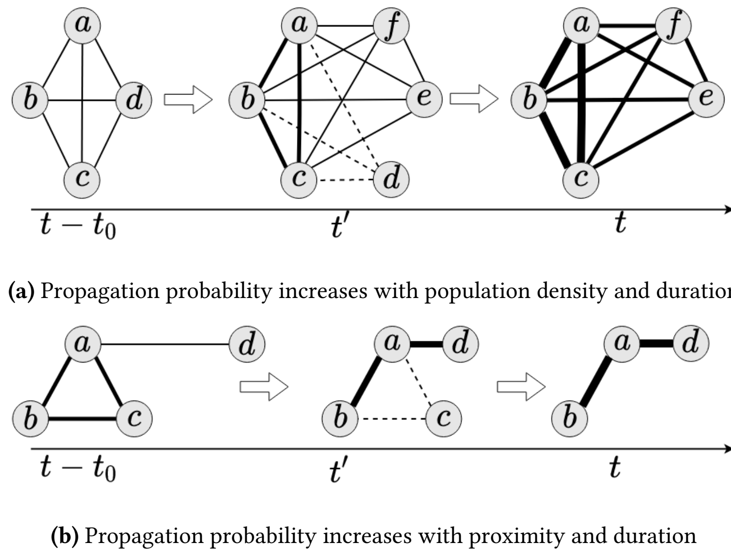
Temporal network

Since the standard IC model uses a static probability distribution over a static network, it is insufficient to handle evolving graphs with changing propagation rates. In real-world settings, both the structure of $\mathcal{G}$ and the propagation rates may change dynamically. For example, the spread of rumours depends on the dynamics as well as the duration of contacts between individuals, and is guided by the specific short-lived nature of gossip or fake news. Similarly, a disease spread model needs to consider the order and duration of interactions within a population, and the varying infectivity of a virus over its lifetime. Therefore, a new temporal IC model must be defined for a temporal network.

Definition 4.1.1 (Temporal Network). *Given a discrete time window $T = \{1, 2, ..., n\}$, a temporal network is a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, P(t))$ where for each time interval $t \in T$, edge set $\mathcal{E}$ is associated with a different propagation probability distribution $P(t)$: $\mathcal{E} \mapsto [0, 1]$. That is, each edge $(u, v) \in \mathcal{E}$ has n probabilities $\wp(u, v, t)$, one for each $t \in T$.*

Since u and v may be linked multiple times in T, the corresponding $\wp(u, v, t)$ for every interval t needs to be separately maintained. An evolving graph is represented by adding all edges and assigning $\wp(u, v, t) = 0$ when there is no (u, v) connection in interval t. Moreover, a rigorous formulation for $\wp(u, v, t)$ is needed to describe more complicated cases of spread.

Figure 4.1 illustrates the impact of temporal order and dynamic connections in a spread model on a sample location-based interaction network (e.g., to track the spread of disease). Suppose node (c) is initially active. In the first case, as shown in Figure 4.1(a), nodes (a) and



(a) Propagation probability increases with population density and duration

(b) Propagation probability increases with proximity and duration

Figure 4.1: Example of dynamic propagation probabilities in the network that reflect temporal characteristics. Edge connection indicates proximity, with thicker edge for higher probability (based on population density, proximity, and duration of contact) and dotted edge for latent probability after deletion.

(b) have higher likelihood of activation than nodes that arrive later, such as (e) or (f), or nodes that leave, such as (d). The greater chance of activation due to the prolonged duration of contact with the active node (c), as well as the increased risk of activation due to the high density of nodes, is reflected in the thicker edge connections to (a) and (b). In the second case, shown in Figure 4.1(b), (a) and (b) are again more susceptible due to their proximity to (c), because (c) leaves before (d) approaches closer. The risk of activation for (d) then increases as the node approaches closer to other nodes that may have been activated by (c).

4.2 Temporal Independent Cascade (T-IC) model

Use of a temporal IC model can help to ascertain if and how spread within a network occurs, by considering evolving network topology alongside granular contact/interaction information to model spread and build solution sets.

T-IC model properties desired

There are a number of challenges that existing solutions do not handle effectively all at once, namely: evolving connectivity patterns, dynamic propagation patterns, approximation guarantees, and sentinel/susceptible solution sets.

i) *Evolving connectivity patterns*: The spread model must cater for both the addition/deletion of edges and varying contact frequencies. For example, connectivity patterns such as the interactions between infected individuals over time [91], or the usage of removable storage devices in a computer network affected by a computer virus [199], can govern how the spread takes place. Previous cascade models that operate on such evolving connectivity patterns (and preserve approximation

guarantees) often use static snapshots at regular intervals to represent the dynamic nature of the network. Rather than using partially observed connections from an incomplete static snapshot to produce solutions that are more likely to be sub-optimal [122], there is a need for a solution set that is optimised across the entire window of time-varying contact events. Furthermore, it is desirable for the spread model to capture temporal dependencies at every step.

ii) *Dynamic propagation patterns*: There is currently no comprehensive solution with provable quality guarantees to handle dynamic propagation at different rates, where the likelihood of the spread of activation along connections can exhibit arbitrary temporally evolving patterns. For example, the bounded infectious period of a disease-causing virus [91] can be captured by a dynamic propagation rate that tapers down to 0 after a defined duration of time. Similarly, dynamic propagation patterns can be used to model the limited period of effectiveness of a rumour [59], or the more long-term effects on a machine infected with a computer virus [199].

iii) *Approximation guarantees*: Prior IC based solutions lose approximation guarantees when modified to support time windows or graph snapshots. To allow for scalable analysis that can support the use of fine-grained temporally evolving graph data, rigorous approximation guarantees must be maintained in the temporal spread modelling algorithm. However, providing such solution quality guarantees should not come at the cost of excessive running time.

iv) *Sentinel/susceptible solution sets*: While cascade models have been predominantly considered for influence maximisation and identifying highly influential nodes, they can also be enhanced to identify i) *sentinel nodes* that quickly detect activation anywhere in the network, or ii) *susceptible nodes* that easily collect activation from anywhere in the network. A set of sentinel (or detector) nodes is one that provides the best coverage over the entire network. That is, spread processes taking place anywhere in the network are likely to reach (and be detected by) one of these sentinels. Identifying such a sentinel set involves an optimisation objective that is different from that of influence maximisation, and requires a novel 'reverse spread' process. Another interesting objective is the identification of high priority susceptible nodes, i.e. those that are independently most likely to collect activation. For such a solution set, the overall network coverage is disregarded in favour of individual risk.

T-IC model formulation

To address the above challenges, the T-IC model is defined over an evolving network. A temporal network has already been formalised in Definition 4.1.1 as one where the propagation probability between two nodes can vary over time. The T-IC model handles dynamic propagation rates and allows an active node to repeatedly try to activate its neighbours. That is, while spread may continue to take place as long as a node is active, the actual likelihood is determined by the propagation rate (e.g., it may drop to zero quickly depending on the infectious period of a virus [91], or based on the short-term nature of rumours [59]). This is an enhancement of the well-known IC model, as it takes the temporal ordering of activations

into account and allows for dynamic changes in both propagation probability and network connectivity patterns.

Given time intervals $i, j \in T$ such that $i < j$, the T-IC model proceeds from i to j as follows: Let A_t denote the set of initially active nodes at the beginning of time interval t. Within each interval $t \in [i, j]$, the standard IC model is executed once under probability distribution $P(t)$ on edges. That is, the active nodes for that particular interval activate their neighbours based on the propagation rates $\wp(u, v, t)$ associated with the edges for that time interval only. This proceeds until no further spread is possible. Note that this $\wp(u, v, t)$ varies, and can fall to 0 to indicate that spread does not take place at the given t, thereby allowing the model to closely replicate real-world dynamic spread. The set of all activated nodes at the end of interval t is A_{t+1}, which thus also represents the active nodes at the beginning of the next time interval $t+1$ (active nodes remain active in subsequent intervals). This process is executed for each interval $i, i+1, \ldots, j$ and the final set of active nodes A_{j+1} is obtained after interval j.

In other words, the standard IC process is unmodified and run to completion independently within each discrete time interval $t \in [i, j]$ under the corresponding probability distribution defined over edges. For the entire chosen time window ($[i, j] \in T$), stacking several of these IC processes, and treating activated nodes as the seeds for the next run, allows the spread over an evolving graph to be simulated without sacrificing the approximation guarantees. Furthermore, if a node is activated in a specific time interval, it can continue to activate its neighbours in subsequent time intervals, but subject to the propagation probability distribution. For example, a person infected with a virus may continue to spread infection during interactions with people repeatedly, but the probability rate remains high only for as long as they are contagious and then drops to 0. This continuous activation during the T-IC process, along with the propagation probability formulations, makes it possible to reflect real-world spreading phenomena (e.g., for disease monitoring) with the T-IC model while maintaining the solution quality guarantees.

4.3 Hypergraph-based reachable set construction

Now that the T-IC process is introduced for detection and analysis of spread (e.g., disease outbreaks, misinformation campaigns), its spread function and hypergraph-based sampling and construction strategy to determine random reachable sets are discussed in greater detail.

Submodular spread function

While other models lose the rigorous approximation guarantees with changes to support temporal spread, the significant contribution presented here enhances a well-defined sampling approach and allows optimality guarantees to be preserved in a new dynamic setting. The defined reverse spread function $\phi(\cdot)$ is first proven to be submodular in Theorem 4.3.1. It follows that a standard hill-climbing greedy algorithm using this spread function achieves a $1-\frac{1}{e}$-approximation guarantee, i.e., a solution that uses this function can be approximated to within $1-\frac{1}{e}$ of optimal [86]. Thereby, the novel spread function utilises dynamic propagation

probabilities for every edge in the network while preserving solution quality unlike many popular temporal IC solutions.

Theorem 4.3.1. *Under the T-IC model, function $\phi_{ij}(\cdot)$ is submodular.*

Proof. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \wp)$ be a directed graph where each edge $(u, v) \in \mathcal{E}$ is associated with a weight $\wp(u, v)$ denoting the probability that spread occurs from u to v. Kempe et al. [86] showed that the IC model induces a distribution over graph $\mathcal{G}$, such that a directed graph $g = (\mathcal{V}, \mathcal{E}')$ can be generated from $\mathcal{G}$ by independently realizing each edge $(u, v) \in \mathcal{E}$ with probability $\wp(u, v)$ in $\mathcal{E}'$. In a realised graph $g \sim \mathcal{G}$, nodes reachable by a directed path from a node u are its reachable set $R(u, g)$, and correspond to the nodes activated in one instance of the IC process with u as the initially active seed node. They proved that for $S \subset V$, the spread function $\sigma(S, g) = \left| \bigcup_{u \in S} R(u, g) \right|$ is submodular.

Similarly, the T-IC model induces a distribution over $\mathcal{G} = (\mathcal{V}, \mathcal{E}, P(t))$, where the IC model is executed independently in each discrete time interval $t \in [i, j]$ under the corresponding probability distribution defined over edges. Additionally, an activated node remains active in subsequent intervals, getting multiple chances to activate its neighbours. So, directed graph $g_{ij} = (\mathcal{V}, \mathcal{E}')$ can be generated as follows: For intervals $t = i, i+1, \ldots, j$, each edge $(u, v) \in \mathcal{E}$ is realised in $\mathcal{E}'$ with probability $\wp(u, v, t)$, only if node u is active at the beginning of interval t. Hence, the reachable set $R(s, g_{ij})$ corresponding to a node s on the generated graph g_{ij} consists of all the nodes that are reachable and activated by time interval j by the seed s that was initially active in time interval i.

Let g_{ij}^T denote the transpose of g_{ij}, obtained by reversing all its directed edges. Reachable set $R(r, g_{ij}^T)$ corresponds to all seed nodes that, if active in interval i, would have the ability to activate the receiving node r by time interval j. Given a set of nodes S, let the reverse spread $\phi(S, g_{ij})$ denote the number of nodes that can reach some node in S. That is, $\phi(S, g_{ij}) = \left| \bigcup_{u \in S} R(u, g_{ij}^T) \right|$. Since $\phi(S, g_{ij}) = \sigma(S, g_{ij}^T)$, the submodularity of $\phi(S, g_{ij})$ follows. Therefore, the expected reverse spread $\phi_{ij}(G, S) = E(\phi(S, g_{ij}))$ is submodular, being a linear combination of submodular functions. □

Hypergraph-based sampling strategy

Borgs et al. [19] use a state-of-the-art sampling strategy to build a hypergraph representation and estimate the spread of activation. In this section, an enhancement of this technique is presented to handle dynamic propagation rates and identify solution sets for both the newly defined optimisation tasks (i.e., identifying sentinel nodes and susceptible nodes). The algorithm and sampling strategy use a novel process of generating the hypergraph to encode the reverse spread of any given subset of nodes via its nets. A hypergraph is a generalisation of a graph in which any number of nodes may be connected by a hyperedge (net). The nodes contained in a net are called its pins. The two-step sampling strategy is as follows: i) random T-IC processes (that start with random active seeds) are executed on the temporal network, and ii) for each execution of a T-IC process, a hypergraph net is constructed whose pins are the nodes that are activated during the process.

As shown in Theorem 4.3.1, g_{ij} can be drawn from the distribution induced by a T-IC model on G. The edges in the graph g_{ij} are tried to be realised by traversing only live edges (i.e., edges where the starting node is already active). This constraint of only considering live edges means that the construction encodes the dynamic nature of the spread, because sampling takes place by respecting the temporal ordering of connections. If a node v is reachable from many different nodes in g_{ij}, then it is more likely that this node will be activated by time interval j. Since any random seed in time interval i is equally likely to start the spread, the existence of more paths that lead to the node v results in a higher likelihood of its activation. This means that the reachable set of nodes $R(u, g)$ (i.e., all the nodes from the realised graph g_{ij} that are reachable by a directed path of edges from the node u), which depends on the random seed node u, is one among many possible sets of activated nodes at the end of a random T-IC process on the randomly sampled g_{ij}. Note that since traversal over live edges helps to capture temporal dependencies in the spread model, the identification of sentinel/susceptible nodes is a non-trivial and orthogonal problem that cannot be achieved through traditional influence maximisation.

Random reachable sets

Overall, the solution depends on two levels of randomness that are encountered during the hypergraph construction: i) the sampling strategy for $g_{ij} \sim \mathcal{G}$, and ii) the computation of $R(u, g_{ij})$ given a random seed u. The former depends on the probability distribution induced by the T-IC model over $\mathcal{G}$, while the latter depends on the seed node u. Such a reachable set that is generated by two levels of randomness is referred to here as a 'random reachable set' $RR(u, g_{ij})$. Outbreaks are typically thought to start from a single source [72, 90]. Therefore, one random seed node is considered in the simulations.

The main sampling step is repeatedly performed to build a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ where each net $n_u \in \mathcal{N}$ is independently generated by executing a random T-IC process from seed u. The hypergraph corresponds to a random reachable set $RR(u, g_{ij})$, i.e., $\text{pins}(n_u) = RR(u, g_{ij})$. The solution quality and concentration bounds thus depend upon the number of nets generated to build the hypergraph [19].

Note that $\mathcal{H}$ and $\mathcal{G}$ are composed of the same set of nodes $\mathcal{V}$. For a solution set S, the number of nodes sharing a net with at least one node in set S (which are referred to as $\deg(S)$ henceforth) corresponds to the number of times a node in S gets activated during the random T-IC processes executed to compute the random reachable sets. To select S as a collection of sentinel nodes, higher $\deg(S)$ will be more likely to detect spread in the network, which can be understood as follows: The degree of a node in the hypergraph is the sum of $|\mathcal{N}|$ Bernoulli random variables [19]. This is because the inclusion of a node v in a random reachable set $RR(u, g_{ij})$ and in $\text{pins}(n_u)$ can be considered as a Bernoulli trial with success probability $\mathscr{p}_v$, where $\mathscr{p}_v$ denotes the probability that v gets activated in a random T-IC process. That is, the hypergraph node degrees are binomially distributed with an expected value $\mathbb{E}[deg(v)] = \mathscr{p}_v \times |\mathcal{N}|$. This implies that $\mathscr{p}_v = \mathbb{E}[deg(v)/|\mathcal{N}|]$. Since the reverse spread can be written as $\phi_{ij}(v) = |\mathcal{V}| \times \mathscr{p}_v$, the node degree corresponds to an estimation of reverse spread of node v. Similarly, in hypergraph $\mathcal{H}$, the expected value $\mathbb{E}[deg(S)/|\mathcal{N}|]$ corresponds to the probability that *at least one* node in S gets activated

during a random T-IC process. Therefore, the degree of a set S of nodes in hypergraph $\mathcal{H}$, corresponds to an estimation of the reverse spread $\phi_{ij}(S) = |\mathcal{V}| \times \mathbb{E}[deg(S)/|\mathcal{N}|]$, which can be estimated well if a sufficient number of nets are built.

4.4 Optimisation objectives

Two distinct objectives and associated T-IC based solutions are introduced in this context: i) finding sentinel nodes, and ii) finding susceptible nodes. These are applicable to spread monitoring in evolving networks to i) detect *if* there is any spread by checking a limited number of nodes, and ii) understand *how* it is likely to spread by identifying the most susceptible nodes.

Novel spread functions are defined to address these two objectives. The above efficient hypergraph-based sampling strategy captures evolving connections and dynamic propagation rates in the network, and T-IC processes generate 'random reachable sets' that can reflect the patterns of spread within this evolving network. These random reachable sets are used to develop two solutions, *Reverse Spread Maximisation (RSM)* and *Expected Spread Maximisation (ESM)*, that can be used to identify sentinel and susceptible nodes respectively, with provable approximation guarantees for the optimality of these solution sets.

RSM is employed to find sentinel nodes, i.e., a set of nodes where at least one is likely to be activated regardless of where the spread begins. RSM is useful to detect *if* there is spread within a population. An application is 'sentinel surveillance' and the early detection of outbreaks by using sensors at the set of sentinels [12, 35, 73]. To understand *how* the spread is likely to take place, ESM is studied to identify susceptible nodes, i.e., a set of nodes that accumulate the most spread from arbitrary seeds. A susceptible set represents all individuals most liable to be part of the spread.

Sentinel nodes

The first objective is to identify the sentinel nodes, i.e., the set of nodes that maximises the probability of detecting any activations in a network where the set of initially active nodes is not known. In other words, this optimisation objective focuses on finding k-element subsets that can behave as sentinels. The proposed solution can also be run until termination over an indefinite time window, but when there is a temporal constraint within which to detect outbreaks, it is noted that a truncated solution set (choosing only k nodes) can sufficiently cover the entire relevant spread.

The objective is to maximise the probability that at least one node in a k-element solution set S becomes active after a random T-IC process (i.e., one that starts with randomly selected seeds) within a given time window $[i, j]$. The 'temporal reverse spread maximisation' objective, as defined below, corresponds to this goal of identifying a k-element set of sentinel nodes. Optimising the success rate of detection of spread anywhere in the network by using sentinels can be achieved by maximising the expected amount of 'reverse spread' $\phi(\cdot)$. Expected reverse spread can be defined as the expected number of nodes that can spread activation to the nodes in S. Therefore the expected reverse spread of set S on $\mathcal{G}$ within $[i, j]$, denoted by $\phi_{ij}(\mathcal{G}, S)$, is the expected number of nodes that can activate at least one node

in set S during a random T-IC process in $[i, j]$. The problem of maximising the expected reverse spread $\phi_{ij}(\mathcal{G}, S)$ can be formally defined as follows:

Definition 4.4.1 (Temporal Reverse Spread Maximisation)**.** *Find the k-element subset of nodes $S^* \subset \mathcal{V}$ such that*

$$S^* = \underset{S \subset \mathcal{V}, |S| = k}{\arg\max} \ \phi_{ij}(\mathcal{G}, S) \tag{4.1}$$

A practical application of finding this optimal solution is to detect an outbreak using minimal resources (e.g., medical tests, computer virus checks, or online content reviews). Testing a targeted set of individuals can be an efficient way to detect the onset of spread within a population before it is widespread, akin to the aforementioned concept of sentinel surveillance.

Susceptible nodes

To understand *how* the spread takes place within the network, it is instead necessary to identify the high priority nodes that collect activation, i.e., nodes that are the most susceptible to activation.

Hence, this aim is to identify the k-element subset S containing the maximum expected number of active nodes after T-IC process in $[i, j]$. Similarly to the temporal reverse spread maximisation objective, the 'temporal expected spread maximisation' objective defined below corresponds to the goal of identifying a k-element set of susceptible nodes. This second problem is formally defined as follows:

Definition 4.4.2 (Temporal Expected Spread Maximisation)**.** *Let $I(s)$ be an indicator random variable for node $s \in S$ such that*

$$I(s) = \begin{cases} 1 & \text{if } s \text{ is activated} \\ 0 & \text{if } s \text{ is not activated} \end{cases} \tag{4.2}$$

after executing random T-IC process. Find a k-element subset of nodes $S^* \subset \mathcal{V}$ that maximises the expected number of active nodes in S^*

$$S^* = \underset{S \subset \mathcal{V}, |S| = k}{\arg\max} \ \mathbb{E}\left[\sum_{s \in S} I(s)\right] \tag{4.3}$$

For example, in disease, computer virus, or misinformation monitoring applications, this objective helps to identify the subset S that are all highly likely to be infected. These offer important candidates to immunise, disconnect, or re-educate in order to mitigate the spread.

4.5 RSM and ESM solution algorithms

This section describes the two algorithms, *Reverse Spread Maximisation* (RSM) and *Expected Spread Maximisation* (ESM), proposed to efficiently compute solution sets for the tasks corresponding to Definitions 4.4.1 and 4.4.2 respectively.

Reverse Spread Maximisation solution

In the hypergraph $\mathcal{H}=(\mathcal{V},\mathcal{N})$, if a node connects many nets (i.e., its degree is high), then that node has a high probability of being activated during a random T-IC process. Similarly, if a set S of nodes covers many of the nets (random reachable sets), then its expected reverse spread $\phi_{ij}(\mathcal{G},S)$ is likely to be higher. In other words, there is a larger set of nodes that all have a chance to activate at least one node of S within the time window $[i,j]$.

As in the maximum coverage problem, the goal is to cover the maximum number of nets (elements) in the hypergraph $\mathcal{H}$ by choosing a solution set S of $\mathscr{k}$ nodes (subsets). This step is therefore equivalent to the well-known NP-Hard maximum coverage problem [157]. Borgs et al. [19] show that the maximum set cover computed by the greedy algorithm on the hypergraph yields $(1-\frac{1}{e}-\epsilon)$-approximation guarantee for influence maximisation. Here, the parameter ϵ relates the approximation guarantee to the running time of the algorithm, and the solution quality improves with the increasing number of nets in the hypergraph.

Algorithm 4.1: RSM Solution

```
input: G = (V, E, P(t)), i, j, k, |N|
1  H = (V, N = ∅)
2  S = ∅
3  for n = 1 to |N| do
4  |  Select source node s ∈ V uniformly at random
5  |  A = {s}
6  |  for t = i to j do
7  |  |  BFS_Q = A
8  |  |  while BFS_Q ≠ ∅ do
9  |  |  |  u = dequeue(BFS_Q)
10 |  |  |  foreach (u, v) ∈ E do
11 |  |  |  |  Draw p ∈ [0, 1] uniformly at random
12 |  |  |  |  if p ≤ p(u, v, t) and v ∉ A then
13 |  |  |  |  |  A = A ∪ {v}
14 |  |  |  |  |  enqueue(BFS_Q, v)
15 |  N = N ∪ {A}
16 for k = 1 to k do
17 |  v_k = arg max_v deg_H(v)
18 |  S = S ∪ {v_k}
19 |  Remove v_k and all of its incident nets from H
20 return S
```

Algorithm 4.1 displays the overall execution of the proposed solution. It generates a number of random reachable sets by first drawing a graph g_{ij} from the distribution induced by T-IC model on the input graph $\mathcal{G}$ and then performing a breadth-first search (BFS) starting from a randomly selected node u. This randomised BFS through time intervals proceeds such that the set of source nodes at each interval are the activated nodes in the preceding interval (lines 6–14). Thus each edge $(u,v)\in\mathcal{E}$ is searched with probability $p(u,v,t)$ in time interval t. All nodes activated during a random BFS form a random reachable set and are connected by a net in hypergraph $\mathcal{H}$ (line 15). After generating the hypergraph $\mathcal{H}$ with $|\mathcal{N}|$ nets, the algorithm repeatedly chooses the highest degree node at each iteration, adds it to the solution set, and subtracts this node together with all incident nets from the hypergraph.

This is done repeatedly until a k-element subset of nodes, which is the resulting solution set S, is computed (lines 16–19). This algorithm generates a solution of sentinel nodes for Definition 4.4.1.

Expected Spread Maximisation solution

In order to maximise the expected number of active nodes in S, all the nodes having the highest probability of being activated should be included in the solution set, since the expected number can be given as $\mathbb{E}\left[\sum_{s\in S} I(s)\right] = \sum_{s\in S} p_s$.

Algorithm 4.2: ESM Solution

input: $\mathcal{G} = (\mathcal{V}, \mathcal{E}, P(t))$, i, j, k, $|\mathcal{N}|$

```
1  H = (V, N = ∅)
2  S = ∅
3  for n = 1 to |N| do
4      Select source node s ∈ V uniformly at random
5      A = {s}
6      for t = i to j do
7          BFS_Q = A
8          while BFS_Q ≠ ∅ do
9              u = dequeue(BFS_Q)
10             foreach (u, v) ∈ E do
11                 Draw p ∈ [0, 1] uniformly at random
12                 if p ≤ p(u, v, t) and v ∉ A then
13                     A = A ∪ {v}
14                     enqueue(BFS_Q, v)
15     N = N ∪ {A}
16 S = argmax_{V' ⊂ V, |V'| = k} Σ_{v ∈ V'} deg_H(v)
17 return S
```

Hence, the problem in Definition 4.4.2 can be solved by Algorithm 4.2. The final step (line 16) now selects the solution set as the k-element subset of nodes having the most incident nets (i.e. largest degrees in $\mathcal{H}$).

As illustrated in Figure 4.3, the proposed sampling strategy is applied on g_{ij}, allowing solutions for the objectives of finding sentinel/susceptible nodes. Clearly, the sampling step is dependent on temporal dynamics which change upon transposing g_{ij} (Figure 4.2), therefore solutions for these two objectives are distinct from influence maximisation approaches in literature. Specifically, the RSM approach can detect any outbreak efficiently, e.g., sentinel nodes (b) and (e) (or (b) and (f)) collectively correspond to greater coverage of the network than that offered by the ESM solution (nodes (b) and (a)). The ESM solution appears in RR-sets more frequently, and are nodes that are all highly likely to be activated. Such sentinel nodes are relevant in settings such as disease monitoring (for contact tracing efforts), or computer virus tracking (to identify and restore infected machines quickly). Identifying susceptible nodes in a social network also has interesting use cases such as combating rumour spread and misinformation campaigns more effectively.

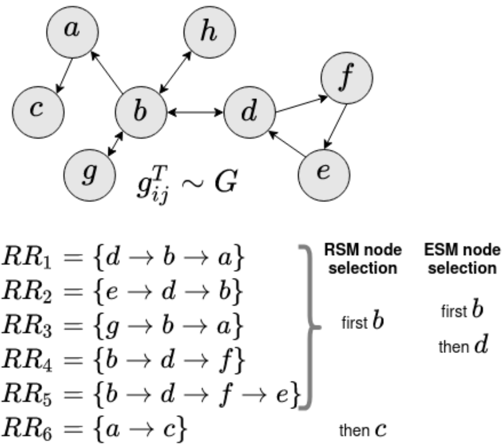


Figure 4.2: RSM vs ESM on g_{ij}^T

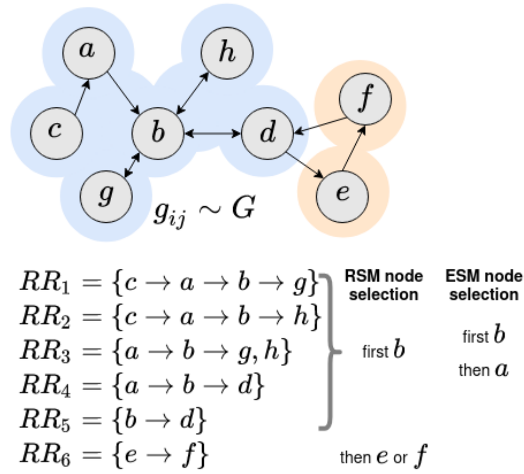


Figure 4.3: RSM vs ESM on g_{ij}

4.6 Experimental evaluation

For each real dataset, a temporal network is built on which the T-IC process is executed. The solutions are evaluated in terms of identifying sentinel nodes and susceptible nodes as defined in Section 4.4.

Datasets

Eight real datasets are used to build temporal networks. The T-IC model can be used to analyse a variety of application settings, such as monitoring disease or malware spread, countering misinformation campaigns, and tracking social influence. For an example use case of disease monitoring, six location-based networks are considered. The first two are spatio-temporal network datasets constructed using the locations (check-ins) of Foursquare users, in line with other research studies on disease monitoring applications [17, 165]. These are referred to as **`NYC`** and **`Tokyo`** datasets. The `NYC` and `Tokyo` datasets [175] record check-in times, (anonymised) user IDs, venue IDs, venue locations, and venue categories. The temporal and spatial information from these are used to build edge connections in the

network of users, selecting 25 consecutive days' data. To alleviate sparsity, the nodes for all users visiting the same venue in the same day are connected bidirectionally. Similarly, the third dataset **SP-office** [54] taken from SocioPatterns[1] contains a temporal network of contacts between individuals in an office building, where active contacts are recorded at 20-second intervals. A small interval of 6 hours is considered to determine whether interactions take place between individuals, based on whether their active locations are the same. Since all the location data is collected from departments within the same workplace, using a larger interval would result in a fully connected contact network. Information about departments is provided which is similar to venue categories in the first two datasets. The first 8 consecutive days are considered to construct the temporal network.

The fourth location dataset is based on SafeGraph,[2] which has been used to analyse mobility patterns for COVID-19 mitigation [28]. SafeGraph contains POIs, category information, opening times, as well as aggregate mobility patterns such as the number of visits by day/hour and duration of visits. Using these mobility patterns, synthetic trajectories are generated for 2K individuals visiting 100 unique POIs in the NYC area over 25 days. To build an individual's trajectory, for each day of the week for three consecutive weeks, sequential visit locations are selected and assigned to appropriate timestamps (based on travel time and visit duration) as follows: i) each individual receives a random start timestamp for travel, a random start POI location, and a random trajectory length that determines the number of POIs to visit, ii) SafeGraph dwell time estimates and a random distance-based travel time are used to determine the timestamp for reaching the next location, iii) depending on this timestamp, POIs are filtered out from the candidate list (based on opening time, category information, and distance from current location) to ensure that the trajectory sequences generated are feasible and realistic, and iv) the next location POI is selected from among the remaining candidates, and the process (steps ii–iv) is repeated until the full length trajectory is complete (where no candidates exist, the trajectory is truncated). Then, the corresponding contact network is constructed by connecting (bidirectionally) nodes that appear in the same location at the same time, considering 5 minute intervals to determine this overlap. This semi-synthetic network is called **SG-traj**.

Analysis of spread on such networks has been widely studied and provides relevant baselines for comparison. Nonetheless, adjustment of the propagation parameters and model activation state types can reflect many other use-cases from computer virus attacks to rumour spread, for any appropriate dataset that provides granular temporal information.

Studies are also conducted on two social network datasets: **wiki-Vote** [102] and **cit-HepPh** [100]. wiki-Vote consists of user discussions on Wikipedia, with edges between users representing votes. cit-HepPh encodes citation connections between research papers. These datasets reflect the typical network structure for problems such as influence maximisation, and propagation probabilities are assigned according to related literature.

Finally, two location datasets developed specifically for studying pandemics are also used to further study the disease use case: **Haslemere** and **Italy**. The first records meetings between users of the BBC Pandemic Haslemere app over time, including pairwise distances

[1] www.sociopatterns.org
[2] www.safegraph.com

with 5 minute intervals [91]. The second reports temporal aggregated mobility metrics for each day's movement of population between Italian provinces based on smartphone user locations before and during the COVID-19 outbreak over 90 days [128]. The `Italy` dataset also provides transition probabilities between provinces, which are directly used as the propagation probabilities for the T-IC process.

The statistics of the constructed networks are in Table 4.1. This reports the total number of temporal edges constructed, and the maximum degree across the entire time window of the temporal edges.

Table 4.1: Dataset properties

Dataset	#Nodes	#Edges	Max degree
NYC	876	18,270	147
Tokyo	765	102,018	311
SP-office	232	78,249	131
SG-traj	2,000	57,530	56
Haslemere	469	205,662	1,506
Italy	111	235,190	6,808
wiki-Vote	8,297	103,689	1,167
cit-HepPh	34,546	841,798	846

Propagation probability settings

Each directed edge (u, v) is assigned a corresponding probability $\wp(u, v, t)$ of propagation from node u to node v at time t, defined based on the needs of the specific application dataset. For example, tracking disease spread may require this propagation to be based on contact duration and physical proximity, which are not relevant to malware in cyberspace. Factors like the period of transmissibility for diseases versus rumours/misinformation have different thresholds that are determined by experts. T-IC supports a variety of settings simply by adjusting the propagation rate formulation as needed. A few of these possibilities are presented below:

• *Sampled from a distribution:* For social network datasets, the propagation probability is assigned following the common practice in influence modelling studies [32] of using a uniform distribution. The edges of the network are randomly assigned to a discrete time interval in T, and $\wp(u, v, t) \in [0, 0.3]$ is sampled for each edge (u, v).

• *Provided by the data:* The `Italy` dataset directly uses the provided transmission probability between connected nodes. In this dataset, it reflects the meta-population migration patterns between different regions (nodes). Note that this network is of a lower granularity than the individual-level trajectories in the location-based datasets.

• *Obtained from domain experts:* For location-based contact networks and `Haslemere`, a domain-informed probability assignment is utilised in the manner proposed by experts. Based on epidemiological studies [5, 60, 91, 120], the propagation probability $\wp$ of a connection from node u to node v at time interval t is calculated as follows, to incorporate knowledge of virus spreading characteristics:

$$\mathscr{p}(u, v, t) = 1 - exp\left(-\sum_{t'} \mathscr{f}(u, v, t')\right)$$

where $\mathscr{f}(u, v, t)$ denotes the 'force of infection' (the larger the value of $\mathscr{f}(u, v, t)$, the greater is the transmission probability between u and v at time t), and $t' \in (t - t_0, t]$ indicates the relevant duration of time up to the current time interval t. The latter is governed by t_0, the duration for which historic infection force is considered, since the transmission probability is decided by the accumulated infection force over t'. Note that $\mathscr{p}(u, v, t)$ can drop to zero to denote no spread, such as once the individual is no longer infectious.

Therefore, a minimal expression for $\mathscr{f}(u, v, t)$ must consider the distance from u to v and the population density at the venue to determine risk, and is formulated in line with the literature [91] as:

$$\mathscr{f}(u, v, t) = ae^{-0.1 \times \mathscr{d}} + be^{-0.1 \times \mathscr{m}^{-1}}$$

where $\mathscr{d}$ is the distance between u and v at time interval t (based on their location data), $\mathscr{m}$ is the number of people located at the same venue, and a, b are hyper-parameters. To realistically simulate spread, a default value of $a = 0.05$ is used, with $b = 0.05$ (for `NYC`, `SG-traj`, and `SP-office` datasets) or $b = 0.01$ (for `Tokyo` dataset due to its dense connectivity). When $\mathscr{d}$ is greater than some threshold or when the dataset has no such proximity information, the contribution to infection force is assumed to be zero (i.e., $a = 0$).

Baselines

There is a lack of research examining sentinel and susceptible nodes on temporal networks, or using IC models. Comparable alternatives to the **RSM** and **ESM** solution sets are thus sought out. Baselines are selected in three groupings. The first consists of IC model-based methods (**Greedy-IM** [86], **DIA** [122]), to demonstrate the superiority of T-IC for analysing spread on evolving networks. The second is the traditional virus propagation method for finding the critical $\mathscr{k}$ nodes to immunise to prevent an epidemic (**T-Immu** [131]). The final grouping covers simple heuristic-based methods (**Max-Deg**, **Random**).

Greedy-IM obtains the top-$\mathscr{k}$ influential nodes using a greedy hill-climbing algorithm over $|T|$ time windows. Since it is infeasible for larger datasets, it is run only on the smallest `Haslemere` and `Italy` datasets. Greedy-IM is applied for each time window separately to calculate the corresponding spread over T and the results are averaged to select the best node. The Dynamic Influence Analysis (DIA) method designs a dynamic index data structure to perform influence analysis over evolving networks. The updating index structure only shows the graph connection at the latest timestamp. Top-$\mathscr{k}$ influential nodes are selected at each time window using DIA, and average results are reported over the $|T|$ time windows. T-Immu formulates a non-linear dynamic system to remove a small set of nodes to prevent an epidemic. An epidemic threshold is also derived for evolving graphs. Both DIA and T-Immu handle temporal and topological information in contact networks, therefore they are run on the location datasets that include such information. The Max-Deg algorithm selects the top-$\mathscr{k}$ nodes in decreasing degree order. The Random algorithm selects $\mathscr{k}$ nodes uniformly at random in a given graph, with average results presented after 20 simulations.

Table 4.2: Normalised performance (reverse spread and binary success rate) at $|T|=25$ with different sizes of solution set $|S|=k$

Dataset	Method	Reverse Spread					Binary Success Rate				
k		10	20	30	40	50	10	20	30	40	50
NYC	RSM	**7.9**	**8.5**	**9.0**	**9.5**	**10**	**8.6**	**7.6**	**9.1**	**8.4**	**10**
	T-Immu	5.0	8.2	8.3	8.7	9.2	7.4	7.1	7.5	6.9	8.6
	DIA	0.0	1.0	2.4	3.2	4.1	0.0	1.1	3.1	4.2	4.8
	Max-Deg	7.5	8.0	8.4	8.8	9.2	7.5	6.7	7.4	6.8	8.3
	Random	4.0	6.3	7.9	8.7	9.2	2.1	6.0	6.6	5.7	6.6
Tokyo	RSM	**8.4**	**8.9**	**9.3**	**9.7**	**10**	**8.4**	**9.1**	**9.0**	**9.2**	**10**
	T-Immu	8.3	8.8	9.1	9.3	9.7	7.9	8.4	8.2	8.5	8.9
	DIA	0.0	0.9	2.0	3.0	4.0	0.0	0.7	1.4	2.7	4.0
	Max-Deg	7.9	8.5	9.0	9.3	9.7	7.6	8.1	7.8	8.4	9.0
	Random	6.6	8.1	8.7	9.3	9.7	5.5	6.9	6.5	7.3	7.8
SG-traj	RSM	**3.1**	**5.4**	**7.3**	**8.6**	**10**	**3.1**	**5.3**	**7.0**	**8.2**	**10**
	T-Immu	1.7	3.2	4.8	6.3	7.3	1.7	2.7	3.8	4.0	5.6
	DIA	0.0	0.1	0.1	0.2	0.2	0.1	0.1	0.2	0.2	0.2
	Max-Deg	2.0	3.3	5.1	6.3	7.5	2.1	2.6	4.2	4.2	6.6
	Random	1.7	3.4	4.9	6.5	7.8	0.0	0.1	0.6	0.7	1.4
wiki-Vote	RSM	**6.4**	**7.9**	**8.8**	**9.5**	**10**	**6.6**	**8.5**	**8.6**	**8.8**	**10**
	Max-Deg	0.0	0.8	1.9	2.9	4.4	0.6	1.8	2.8	3.4	4.1
	Random	0.1	1.1	1.8	2.5	3.0	0.0	0.6	1.3	1.7	1.9
cit-HepPh	RSM	**3.5**	**5.7**	**7.5**	**8.8**	**10**	**3.9**	**4.7**	**7.0**	**8.0**	**10**
	Max-Deg	0.1	0.2	0.3	0.6	0.9	0.0	0.2	0.4	0.6	1.0
	Random	0.0	0.0	0.0	0.4	0.5	0.0	0.0	0.0	0.0	0.0

Setup

The algorithms are executed on an Ubuntu 20.04 machine with 16 Intel 3.90 GHz CPUs and 503 GB RAM.

The proposed solution sets are compared with the influential sets from the alternatives with respect to the following performance measures: i) reverse spread from the solution set, ii) average number of activated nodes (expected spread) in the solution set, and iii) binary success rate of detecting spread. The reverse spread $\phi(\cdot)$ is computed as defined in Section 4.3. The binary success rate is the average number of times that there is at least one active node in the solution set during random T-IC processes. Reverse spread is expected to be correlated with binary success, as both relate to the effectiveness of the solution set (sentinel nodes) in covering/detecting spread in the network. The expected spread, computed as the average number of activated nodes in the solution set, represents its susceptibility. Specifically, 1000 random T-IC processes are simulated to activate nodes in the network within T.

Performance with different solution set sizes k

Tables 4.2 and 4.3 summarise the comparative results on the large-scale location datasets and social datasets. Solution sets (S) of sizes $k=10, 20, \ldots, 50$ are considered, with a time window of length $|T|=25$ days. All results are normalised for ease of comparison, i.e., the range of values between the minimum and maximum is mapped to $[0, 10]$ to produce normalised value $x_n = \frac{x - x_{min}}{x_{max} - x_{min}}$, where x is the original value, and x_{min} and x_{max} are the minimum and maximum values across all the methods on the same measure. For example, consider the

Table 4.3: Normalised performance (expected spread) at $|T| = 25$ with different sizes of solution set $|S| = k$

Dataset	Method	Expected Spread				
k		10	20	30	40	50
NYC	ESM	**2.3**	**3.8**	**6.1**	**7.1**	**10**
	T-Immu	0.4	0.7	1.7	2.0	2.3
	DIA	0.0	0.0	0.1	0.2	0.3
	Max-Deg	0.4	0.5	0.9	1.1	1.8
	Random	0.1	0.4	0.7	0.9	1.2
Tokyo	ESM	**2.2**	**4.5**	**5.9**	**7.8**	**10**
	T-Immu	1.1	2.0	3.5	5.0	6.8
	DIA	0.0	0.0	0.1	0.2	0.3
	Max-Deg	0.9	1.7	1.9	2.9	3.6
	Random	0.3	0.9	1.2	1.7	2.3
SG-traj	ESM	**2.1**	**3.9**	**5.9**	**6.9**	**10**
	T-Immu	0.9	1.4	2.2	2.3	3.4
	DIA	0.0	0.1	0.1	0.1	0.1
	Max-Deg	1.0	1.3	2.3	2.5	3.8
	Random	0.0	0.0	0.3	0.4	0.7
wiki-Vote	ESM	**2.8**	**6.7**	**8.4**	**9.6**	**10**
	Max-Deg	0.1	0.3	0.5	0.8	1.4
	Random	0.0	0.1	0.2	0.3	0.3
cit-HepPh	ESM	**3.5**	**5.1**	**6.7**	**8.4**	**10**
	Max-Deg	0.0	0.2	0.3	0.5	0.7
	Random	0.0	0.0	0.0	0.0	0.0

normalised reverse spread on the NYC dataset shown in Table 4.2 . The value 0 for DIA in this section at $k = 10$ means that the reverse spread of DIA with $k = 10$ is minimum among all methods from $k = 10$ to $k = 50$, while the value of 10 for RSM at $k = 50$ denotes that its reverse spread in this configuration is maximum across among all methods and all k.

The set returned by RSM collectively achieves the highest reverse spread coverage in all cases, which increases with increasing k (solution set size). Without prior information about the initial seeds from where activation begins to spread, distributing limited resources (e.g., scarce/expensive wireless sensors or medical tests) to these sentinel nodes (i.e., the nodes selected in S) increases the probability of detecting the spread at an early stage.

By contrast, ESM selects all nodes having the highest probabilities of being activated during a random T-IC process, and thus best captures the largest expected spread out of all methods. ESM outperforms Max-Deg, which is often enforced in reality, by up to 82% on NYC. ESM is thus an effective targeted strategy for identifying the most susceptible nodes (e.g., for contact tracing or treatment).

The binary success rate using RSM is the best for all datasets and k. Comparisons with T-Immu and DIA show that considering temporal properties while also preserving the overall graph structure is vital to select the ideal solution sets. RSM consistently outperforms T-Immu (nearly 2x better on SG-traj) despite having related objectives, since T-Immu cannot capture time-varying transmission probabilities. RSM also drastically outperforms DIA, which is worse than Random, because DIA selects nodes of the evolving network based on an updating index which only remembers the latest probability assignment and fails to

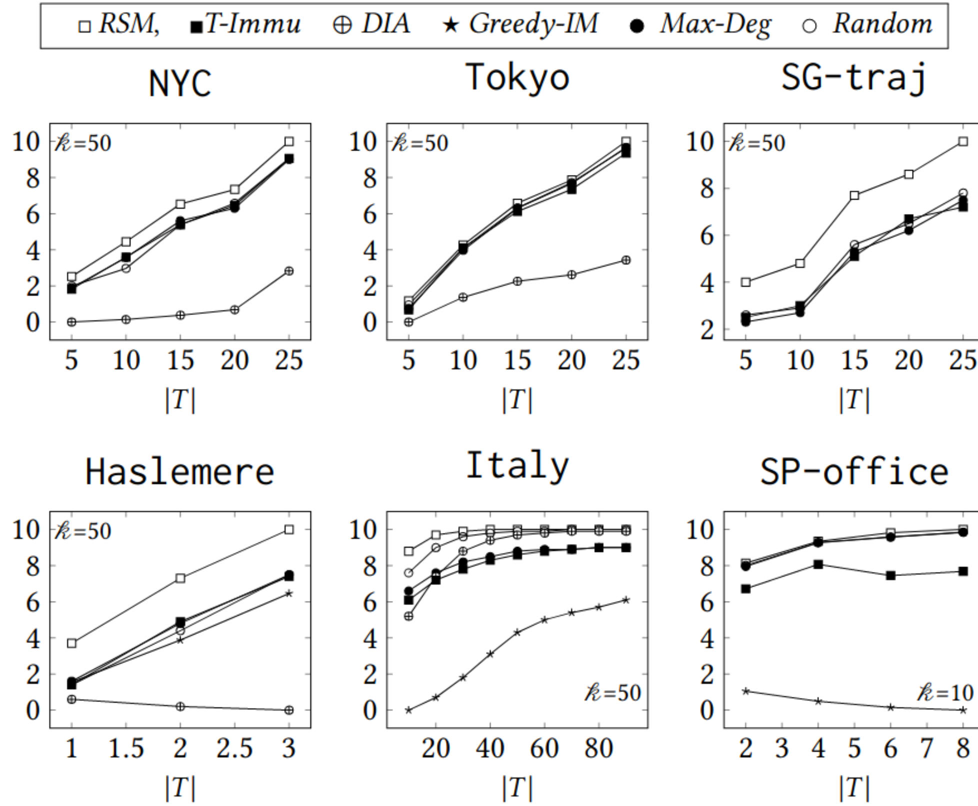


Figure 4.4: Normalised reverse spread with different lengths of time window $|T|$

capture the globally optimal solution set over T. The improvements (at least 10% higher success rates in the worst case) over Max-Deg confirm that the dynamic topology of the network (captured by the RSM and ESM solutions with T-IC process) plays a much more significant role compared to the local connectivity (node degrees) when modelling the spread.

Performance with different time window lengths $|T|$

The effect of varying the time window T is studied while keeping a constant solution set size $k=50$ on `NYC`, `Tokyo`, and `SG-traj` over one-day intervals up to $|T| = 25$. All three datasets provide spatio-temporal trajectories where the evolution of spread over time is relevant. The results are also plotted for `SP-office`, `Haslemere`, and `Italy`. The time window used is different to accommodate these latter datasets. The propagation probability for `Italy` is the real transmission probability of people moving between any two provinces over $|T| = 90$ days, while for `Haslemere` it captures infections over $|T| = 3$ days. For `SP-office`, $|T| = 8$ days is considered, with $k=10$ nodes. Due to the small and densely connected nature of `SP-office`, the baseline algorithms can identify only up to a small number of sentinel nodes. Additional nodes quickly become redundant after already covering the entire contact network. Figures 4.4–4.6 show that reverse spread, expected spread, and binary success all increase with $|T|$, as it allows more activations to take place. As expected, RSM has the best performance with respect to reverse spread (Figure 4.4) and binary success rate (Figure 4.5), and ESM outperforms other methods in terms of having the best expected spread (Figure 4.6).

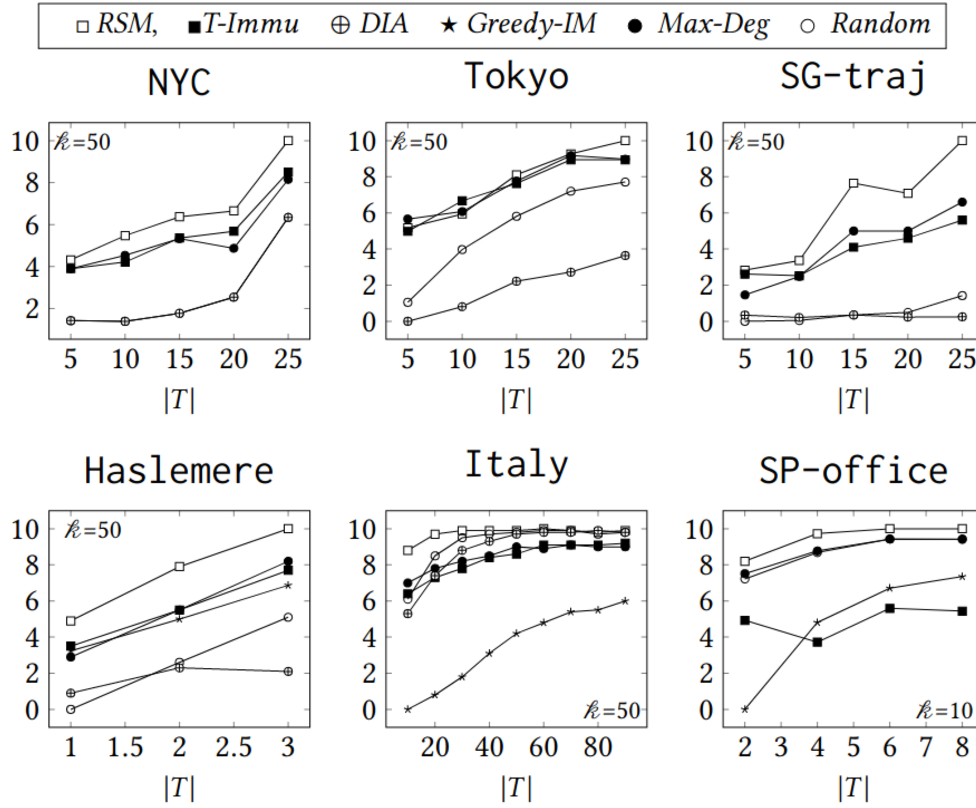


Figure 4.5: Normalised binary success rate with different lengths of time window $|T|$

Only considering the node degrees is ineffective, particularly as propagation becomes more complex, e.g., on a large network and elongated time windows. DIA is jeopardised especially with smaller time windows as the overall optimality of the solution set is not guaranteed by the most recent snapshot of the graph. Specifically, DIA selects the nodes heavily depending on the topology connections. Hence, when the connections are dense across a smaller number of venues (departments), even selecting very few nodes (2 nodes in `SP-office` case) immediately terminates the algorithm. In comparison, this issue can be mitigated in RSM because the propagation process is formulated with a finer granular level. `Haslemere` and `Italy` datasets in these figures also highlight how Greedy-IM cannot effectively capture the optimal global solution over multiple time windows.

The impact of a densely connected network (e.g., `Tokyo`, `SP-office`) on the nature of the spread is visible in Figures 4.4–4.5. Here, selecting a solution set for maximal coverage provides relatively similar results across different methods, although RSM continues to perform best. The average node degree in these networks is high, meaning that activation quickly spreads through the network. This is difficult to combat even considering only $\mathcal{k} = 10$ solution nodes for `SP-office`. While `Haslemere` and `Italy` are also densely connected, the more realistic disease setting of these datasets, and their specific time windows ($|T|$) under consideration, permit the different methods to show more variation in performance.

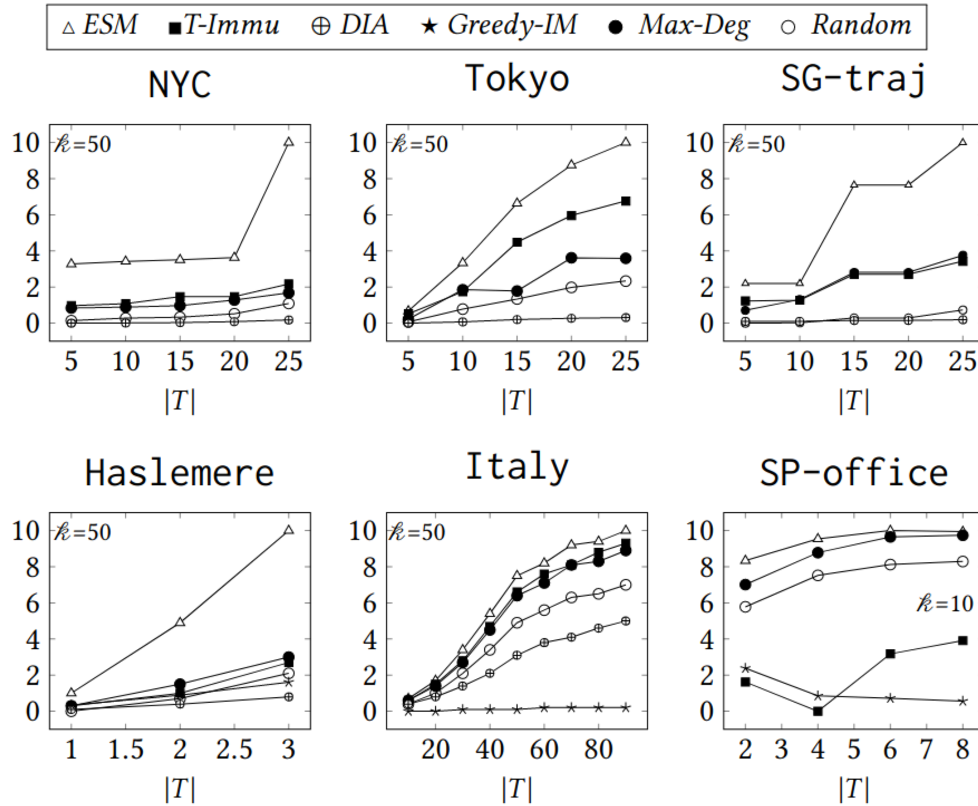


Figure 4.6: Normalised expected spread with different lengths of time window $|T|$

Running time efficiency

While the solution quality improves with higher number of hypergraph nets generated (up to a certain point), there is an efficiency trade-off. The running time of generating hypergraph nets is measured by varying the desired number of nets $|\mathcal{N}|$ and the number of time windows $|T|$ under consideration, as shown in Table 4.4 and Table 4.5, respectively. Specifically, with $|\mathcal{N}|$ increasing from 20K to 100K (for a fixed $|T|=5$) in Table 4.4, there is a slight increase in running time (from 0.43 to 2.18 seconds for `Tokyo`) demonstrating the efficient and scalable nature of the reachable set sampling based algorithm. The hypergraph construction time also increases with $|T|$ (for a fixed $|\mathcal{N}|=20$K) in Table 4.5 due to a prolonged propagation process, which is especially evident in dense networks (e.g., `Tokyo`). Despite the increase in computation time and memory requirements when increasing $|\mathcal{N}|$, stable solution sets of sufficiently high quality are found to be produced without the need for more than 20K nets.

T-IC can model large-scale individual-level contacts efficiently, whereas other solutions [73, 122, 131] are only feasible on small graphs. For example, T-Immu is time-consuming due to repeated computations of the eigenvalue of the dynamic contact network structure which makes it not feasible for large-scale dataset (e.g., `cit-HepPh`), and DIA can only consider the latest snapshot but not the global structure efficiently.

The commonly used IC-based method, Greedy-IM, is used for comparison of the running time for selecting different size of solution set S from $k = 10$ to $k = 50$ in Table 4.6. The running time of Greedy-IM grows quickly and gets infeasible with large dataset and long time

Table 4.4: Running time (in seconds) with different number of nets $|\mathcal{N}|$ (for $|T|=5$)

Dataset \ $\lvert\mathcal{N}\rvert$	20000	40000	60000	80000	100000
`NYC`	0.99	1.95	2.96	4.00	4.99
`Tokyo`	0.43	0.86	1.31	1.75	2.18
`SP-office`	0.83	1.71	2.62	3.51	4.43
`SG-traj`	0.04	0.07	0.11	0.15	0.19
`wiki-Vote`	53.40	108.76	158.52	216.58	271.99
`cit-HepPh`	1.27	2.69	3.98	5.35	6.45

Table 4.5: Running time (in seconds) with different number of time windows $|T|$ (for $|\mathcal{N}|=20\text{K}$)

Dataset \ $\lvert T\rvert$	5	10	15	20	25
`NYC`	0.98	1.80	3.34	5.51	11.32
`Tokyo`	0.42	5.40	22.65	55.94	106.0
`SP-office`	0.83	2.64	6.04	12.21	18.35
`SG-traj`	0.04	0.06	0.10	0.12	0.16
`wiki-Vote`	53.40	149.49	320.87	549.95	917.51
`cit-HepPh`	1.27	7.46	25.68	63.73	130.65

Table 4.6: Running time (in seconds) with different sizes of solution set $|S|=k$ (for $|T|=3$) on `Haslemere`

Method \ k	10	20	30	40	50
RSM	19.15	19.16	19.20	19.23	19.26
ESM	18.34	18.34	18.35	18.35	18.36
Greedy-IM	878.98	2647.99	6729.71	11780.86	18948.14

windows, whereas RSM and ESM running times grow much more slowly. Hence, only a small dataset `Haslemere` is evaluated, and considering a small window of $|T|=3$. While RSM and ESM remain feasible, the running time of Greedy-IM increases sharply from 15 minutes to over 5 hours with larger k, which makes it inapplicable for large evolving networks.

Large-scale datasets are easily supported by the proposed solution. Therefore, several intervention strategies can be efficiently studied using the T-IC process, for simulations on reducing spread in temporal networks. For example, various techniques are applicable to the disease monitoring context, like targeted shutdowns (deleting nodes) or occupancy restrictions (limiting the number of connection edges), backward contact tracing (identifying superspreaders that occur repeatedly among reverse reachable sets), and categorical analysis (node-level study of different categories and their impact on spread). There are analogues to reduce spread in other settings such as with misinformation propagating in social networks.

4.7 Discussion

The Temporal Independent Cascade (T-IC) model is shown to be effective in finding solution sets for the tasks of Reverse Spread Maximisation (RSM) and Expected Spread Maximisation (ESM). These tasks have a wide variety of application settings, from modelling the spread of

diseases in population contact networks to combating misinformation or rumours spreading in social networks.

Reachable sets (nodes that may be affected when the spread occurs) are constructed using a hypergraph-based sampling approach, which is shown to have several benefits in modelling evolving connectivity and spread patterns in dynamic graphs. More crucially, the reverse spread function derived using this T-IC approach is proven to be submodular, allowing the above RSM and ESM algorithms to maintain important approximation guarantees of IC models even when applied to temporal networks. This enables the newly proposed algorithms to efficiently handle large-scale and highly granular data. Unlike the alternatives, the proposed sampling method of T-IC is shown to be scalable, and high quality results are produced within seconds requiring no more than 20K nets constructed.

The hypergraph-based sampling approach only traverses live edges, therefore respecting the temporal dependencies that are imposed by the evolving graph topology. Furthermore, the construction of random reachable sets inherently accounts for the randomness of the seed nodes from where spread first begins. This ensures that, with sufficient number of hypergraph nets being generated, the overall spread in the network (regardless of where it starts) can be modelled with high accuracy.

Two objectives are defined to identify i) a minimal set of sentinel nodes (i.e., nodes which minimally cover the network, for the purpose of spread detection), and ii) a set of highly susceptible nodes (i.e., nodes that are at high risk, for prioritising tracing, intervention, and treatment measures across various use cases). These are notably distinct from the traditional influence maximisation goal, and require novel approaches to identify optimal solution sets. RSM and ESM are respectively shown to be effective for solving these, where nets of the hypergraph are selected to obtain the collective highest network coverage (sentinel nodes) or highest individual activation likelihoods (susceptible nodes).

Through extensive quantitative analysis performed on eight real-world datasets across multiple settings, it is demonstrated that the dynamic topology captured by the T-IC model plays a much more significant role than local connectivity. This is evident in the sentinel nodes identified by RSM having significantly higher success rates of detecting spread compared to T-Immu and DIA. It is also shown that temporal characteristics alongside the global graph structure are needed for optimal solutions, which can be seen as ESM dramatically outperforms Max-Deg as a superior targeted strategy for identifying susceptible nodes.

These findings help to set the stage for further exploration of hypergraph-based optimisation strategies for efficient and distributed computations. In particular, the challenges of temporal networks are examined and tackled here in detail. In the next chapters, hypergraph partitioning models and more hypergraph-based sampling techniques are explored on static graphs, for both graph analytics as well as GNN workloads. Then, the dynamic setting is revisited, with a foray into streaming graph workloads in the final chapter.

Chapter 5

Hypergraph-based Partitioning on Static Graphs for Analytics

> "*He had spent years in* ***search*** *of boredom, but had never achieved it. Just when he thought he had it in his grasp, his life would suddenly become full of near-terminal interest.*"
>
> Terry Pratchett, *Sourcery*

To achieve scalability on large graphs (e.g., by using distributed systems), the common approach is to partition the graph data for parallel processing. Graph partitioners strive to achieve good data locality and balanced workloads across partitions. Communication overheads are involved when partitions need to access data that is not available locally. Hypergraph-based partitioning models have gained popularity, which rely on treating the data/tasks as nodes and using hyperedges to represent the communication requirements between them. In so doing, communication costs incurred from cut nets are more accurately modelled and minimised in the partitioned graph.

Before designing any partitioning or communication scheme, it is important to understand how the use of such a graph/hypergraph partitioning algorithm can affect the system performance and scalability. Methods to partition graphs are typically evaluated based on their performance on running PageRank, connected components, or shortest path algorithms [159]. However, other graph analytics algorithms have their own unique workload patterns, as they may induce different traversals and data accesses. As a consequence, different tasks on partitioned graphs may alter the performance efficiency and the quality of approximate results. This is especially true in a limited communication environment where data locality must be strictly maintained and the machines do not exchange information about their local subgraphs. It motivates the need for bespoke evaluation schemes to assess the efficacy of the partitioning scheme on any new workload.

For example, PageRank [125] is widely used to calculate the importance of a node based on the network structure. The method aggregates scores from the local neighbourhood of each node over repeated iterations. Variations such as personalised PageRank can produce resulting node lists for single-source queries [10, 160]. Other similarity measures have been devised, such as SimRank and RoleSim. Despite commonalities in their overall working, these measures induce very different access patterns in computing their similarity match scores. The former considers if two nodes are similar by checking if all their in-neighbours are similar, while the latter works by averaging only the similarities over the maximum bipartite matching of this in-neighbourhood. These computations are often performed for a single query node instead of for all pairs of nodes over the entire graph. Therefore, running global PageRank may not offer a relevant workload to approximate the workloads obtained when performing single-source SimRank or RoleSim computations. The shortest path algorithm is used with single-source queries, but is still not a good estimate of the above workloads since the entire neighbourhood is not explored when determining the shortest route.

In this chapter, an approximate similarity search workload is considered, and partitioning methods are applied to evaluate the impact on solution quality. Hypergraph-based (and graph-based) partitioning is used to i) scale this model to support large-scale inputs, both without and with strict data locality (i.e., restricted communication across partitions), and ii) evaluate the performance tradeoff in terms of running time and solution quality. The benefits of the hypergraph model over the graph model are demonstrated in this graph analytics setting, where communication across partitions is better reflected by the former to thereby choose more centrally connected separator vertices for caching.

Specifically, RoleSim* [184], a recent similarity model, is employed as the specific use case with real single-source similarity computation query workloads. A partitioning strategy is developed in this chapter to scale RoleSim* on large graphs by taking advantage of its triangular inequality property to prune the search graph. Replicated vertices are used to cache similarity scores, and as a means of communicating with their counterpart on a different part (i.e., when neighbourhoods need to be accessed to compute new similarity scores). Next, this partitioned approximate model is evaluated in terms of ranking quality, efficiency, and scalability using a novel unsupervised semantic evaluation scheme and real RoleSim* query workloads. The results are compared on the partitioned graph to reflect two different settings – one where inter-partition communication is allowed, and one where it is not. These graph and hypergraph partitioning approaches are applicable on any workload. The evaluation scheme designed is useful to determine the best choice of partitioner by testing its quality directly on the target workload, instead of relying on heuristics such as partitioning cut size or replication factor.

5.1 Preliminaries

A vast swathe of graph-based similarity models have been proposed over the years, and the definitions of those studied in this chapter are provided below. Their sparse matrix notation is also discussed to elicit connections to sparsity-based partitioning approaches.

SimRank [80]

SimRank is one of the most widely known similarity measures. It is based on the idea that 'similar nodes are pointed to by similar nodes', i.e., the in-neighbourhoods of similar nodes are recursively similar. SimRank measures the similarity of two nodes based on global graph structure from the paths connecting them. Various types of queries may be performed: single-source, single-pair, all-pairs, and partial-pairs SimRank [6, 99, 103, 110, 182].

Given a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, SimRank uses the global graph structure to measure the similarity between two vertices. Two vertices u and v are similar if their in-neighbours $N_{in}(u)$ and $N_{in}(v)$ are also similar. Formally, the SimRank score is defined as

$$SR(u,v) = \begin{cases} 1, & \text{if } u = v \\ \frac{c}{|N_{in}(u)||N_{in}(v)|} \sum_{a \in N_{in}(u)} \sum_{b \in N_{in}(v)} SR(a,b), & \text{if } u \neq v \end{cases} \tag{5.1}$$

where $c \in [0, 1]$ is a damping factor. The SimRank score between a vertex and itself is always maximum ($SR(u, u) = 1$). The SimRank score of a node with no in-neighbours is always 0.

Despite its popularity, SimRank has disadvantages of producing unreasonable similarity scores in various situations and accommodating only paths of equal length from a common parent node while ignoring many other paths in the similarity estimation.

RoleSim [83]

RoleSim was devised as a role-oriented similarity model, with the intuition that 'two nodes are role similar if they interact with automorphically equivalent sets of in-neighbors'. That is, unlike using the average similarity of all the in-neighbour pairs of (u, v) as in SimRank, in RoleSim only the similarities over the maximum bipartite matching $M(u, v)$ are averaged and the remaining are ignored. RoleSim measures the similarity of two nodes through the paths connecting their different roles, thus even disconnected nodes could be considered similar. Automorphically equivalent nodes, intuitively, are ones that have similar roles. These nodes may exchange roles with minimum effect on the structure of the graph.

Given a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, and two vertices u and v with in-neighbours $N_{in}(u)$ and $N_{in}(v)$, their maximum bipartite matching $M(u, v)$ is given as

$$\begin{aligned} M(u,v) = \Big\{ &(x,y) : x \in N_{in}(u), y \in N_{in}(v), \\ &(x',y') \notin M(u,v), \text{such that } x = x' \text{or } y = y' \Big\} \end{aligned} \tag{5.2}$$

RoleSim then measures the similarity between u and v as

$$RS(u,v) = (1-\beta) \max_{M(u,v)} \frac{\sum_{(a,b) \in M(u,v)} RS(a,b)}{|N_{in}(u)| + |N_{in}(v)| - |M(u,v)|} + \beta \tag{5.3}$$

where $a \in N_{in}(u)$ and $b \in N_{in}(v)$. The parameter $0 < \beta < 1$ is a decay factor.

RoleSim suffers the major drawback of only considering limited information over automorphically equivalent sets and ignoring the rest of the neighbourhood.

RoleSim* [184]

RoleSim* follows a recursive intuition that two similar nodes i) interact with automorphically equivalent sets of in-neighbors and ii) are pointed to by similar nodes out of these sets.

Given a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, and two vertices u and v with in-neighbours $N_{in}(u)$ and $N_{in}(v)$ having maximum bipartite matching $M(u, v)$ as above, RoleSim* then computes the similarity between u and v as

$$\begin{aligned} s(u,v) = \beta\Bigg(\mu \times & \overbrace{\frac{1}{|N_{in}(u)| + |N_{in}(v)| - |M(u,v)|} \sum_{(x,y)\in M(u,v)} s(x,y)}^{\text{Part 1: average similarity over maximum matching } M(u,v)} \\ & + (1-\mu) \times \underbrace{\frac{1}{|N_{in}(u)| \times |N_{in}(v)| - |M(u,v)|} \sum_{(x,y)\in N_{in}\setminus M} s(x,y)}_{\text{Part 2: average similarity over } N_{in}\setminus M = (N_{in}(u)\times N_{in}(v)) - M(u,v)}\Bigg) \\ & + (1-\beta) \end{aligned} \tag{5.4}$$

Here $N_{in}(u) \times N_{in}(v) = \{(x, y) \mid x \in N_{in}(u) \text{ and } y \in N_{in}(v)\}$ are all in-neighbouring pairs of (u, v). This is divided into two subsets $N_{in}(u) \times N_{in}(v) = M(u, v) \cup (N_{in}(u) \times N_{in}(v)) - M(u, v)$, which forms the two parts of the RoleSim* definition. The average similarity over maximum matching $M(u, v)$ in Part 1 provides the contribution from (u, v) interacting with the automorphically equivalent set $M(u, v)$ of in-neighbours. The average similarity over $(N_{in}(u) \times N_{in}(v)) - M(u, v)$ is the contribution from (u, v) being pointed to by the remaining in-neighbourhood pairs from $M(u, v)$. The parameter $0 < \mu < 1$ is the relative weight balancing similarities inside and outside $M_{u,v}$. RoleSim* covers the traditional RoleSim model as a special case when $\mu = 1$, since Equation 5.4 reduces to the traditional RoleSim equation. The behaviour at $\mu = 0.5$ is close to that of the SimRank equation.

The iterative RoleSim* computation between two nodes is shown to be symmetric, bounded, and monotonic. Furthermore, defining $\mathcal{d}(u, v) = 1 - s(u, v)$, for any three nodes a, b, c in a graph, the following triangle inequality holds

$$\mathcal{d}(a, b) + \mathcal{d}(b, c) \geq \mathcal{d}(a, c) \tag{5.5}$$

In other words, it is possible to induce a RoleSim* distance that obeys sum-transitivity in the similarity scores. This property is crucial for the partitioning strategy introduced later.

Sparse matrix notation for similarity measures

A matrix form for the SimRank model is offered by Kusumoto et al. [99], which is equivalent to Equation 5.1. This is defined as follows:

$$\boldsymbol{S}_1 = \max\{c(\boldsymbol{Q}\boldsymbol{S}_1\boldsymbol{Q}^T), \boldsymbol{I}\}$$

where $\boldsymbol{S}_1$ is the SimRank similarity matrix whose entry $\boldsymbol{S}_1(i, j)$ is the SimRank score $SR(i, j)$ and $\boldsymbol{I}$ is an identity matrix. $\boldsymbol{Q}$ is the backward transition matrix and $\boldsymbol{Q}^T$ is its matrix transpose, with $\boldsymbol{Q}$ defined as

$$\boldsymbol{Q}(i, j) = \begin{cases} \frac{1}{|N_{in}(i)|}, & \text{if } \exists (i, j) \in \mathcal{E} \\ 0, & \text{otherwise} \end{cases}$$

The presence of the maximum bipartite matching term makes representing RoleSim and RoleSim* using matrix notation a slightly more involved process. First, a local masking matrix $\boldsymbol{M}$ may be constructed, with entries $\boldsymbol{M}(i, j)$ defined such that

$$\boldsymbol{M}(i, j) = \begin{cases} 1, & \text{if nodes } i \text{ and } j \text{ are maximally matched} \\ 0, & \text{otherwise} \end{cases}$$

Considering $\boldsymbol{S}_2$ as the RoleSim similarity matrix whose entry $\boldsymbol{S}_2(i, j)$ is RoleSim score $RS(i, j)$, it is possible to denote the sum of the RoleSim similarity scores over the maximum weighted bipartite matching as

$$\boldsymbol{S}_2 = \boldsymbol{A}(\boldsymbol{M} \odot \boldsymbol{S}_2)\boldsymbol{A}^T$$

using Hadamard product $\odot$, where $\boldsymbol{A}$ is the adjacency matrix.

This indicates that partitioning strategies are indeed useful to distribute such sparse matrix computations. Hypergraph-based partitioners are particularly well-suited to the task in situations where there is sparsity in the input adjacency matrix. Hypergraph partitioning has been previously applied for parallel PageRank [20], but has not been studied for the similarity search workloads described in this chapter.

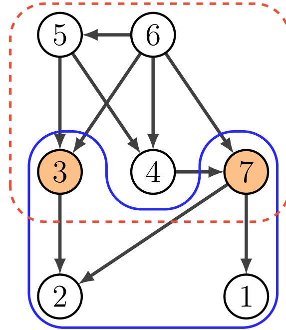
5.2 Partitioning models for scalable similarity search

RoleSim* based similarity search can be scaled via pruning using the triangular inequality property. With some precomputation, it is possible to prune the node-pairs that need not be evaluated and thereby eliminate nodes from the candidate list of most similar nodes. In this section, a strategy is described for retrieving approximate node-pair similarity results in a partitioned graph. Then a method is discussed for exact single-source RoleSim* (SSRS*) computation to obtain the similar nodes to a query q, by indexing based on the distance to some chosen keys.

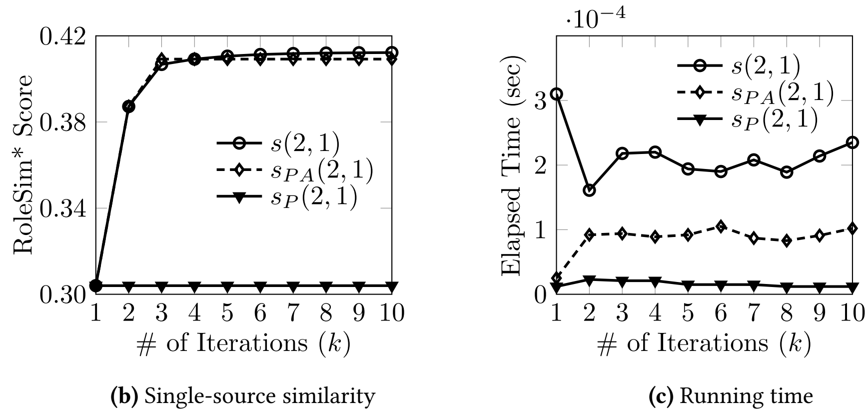
First, a simple partitioning strategy is applied, where graph $\mathcal{G}$ is divided into parts of roughly equal sizes and the important nodes are determined that should be replicated and have their computed similarity scores cached for best efficiency. Identification of these important nodes is done using a vertex replication approach, i.e., vertex-cut partitioning. The set of vertex separators (also called vertex cut or separating set) is referred to as $\mathcal{V}_S$. Nodes in $\mathcal{V}_S$ are the nodes that, when removed from $\mathcal{G}$, separate it into its partitions. This is true for both graph partitioning and hypergraph partitioning, and these vertices are available to all parts in which they are replicated.

Approximate single-source similarity search

Example 5.2.1. Consider the digraph $\mathcal{G}$ in Figure 5.1a, where a 2-way partitioning has been performed resulting in vertex separators $\mathcal{V}_S = \{③, ⑦\}$. Subgraphs $\mathcal{G}_1$ and $\mathcal{G}_2$ are constructed such that $\mathcal{V}_{\mathcal{G}_1} = \{①, ②, ③, ⑦\}$ and $\mathcal{V}_{\mathcal{G}_2} = \{③, ④, ⑤, ⑥, ⑦\}$. Using the SSRS* approach, all pairs of similarities from nodes in $\mathcal{V}_S$ to nodes in $\mathcal{G}$ can be precomputed. Given a decay factor $\beta = 0.8$, and relative weight $\mu = 0.7$, the exact RoleSim* similarities are computed (after k iterations) and cached for all $s(v, *)$ where $v \in \mathcal{V}_S$. Now, for node-pair $(2, 1)$, the exact similarity score of $s(2, 1)$ from $\mathcal{G}_1$ may be approximately computed as $s_P(2, 1)$ where the precomputed value of $s(3, 7)$ is used in every iteration and only the pruned graph $\mathcal{G}_1$ is considered, i.e., strict data locality is enforced without inter-partition communication. The similarity score $s_{PA}(2, 1)$ also uses the precomputed $s(3, 7)$, but also permits inter-partition communication to neighbouring nodes if they are present in $\mathcal{G}_2$. As seen from Figures 5.1b and 5.1c, $s_{PA}(2, 1)$ is more accurate but less efficient due to accessing a larger graph.



(a) Partitioned graph $\mathcal{G}$ with the two partitions encircled in dashed red and solid blue respectively



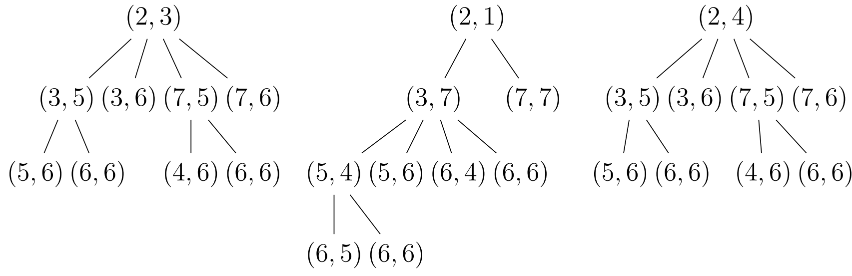
(b) Single-source similarity

(c) Running time

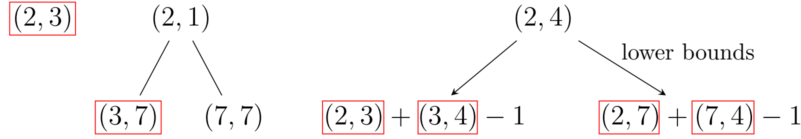
Figure 5.1: Approximate RoleSim* node-pair similarity retrieval performance on graph $\mathcal{G}$

Consider a query node q. To compute its similarity to any node n in $\mathcal{G}$, there are three different cases:

1. One or both nodes are vertex separators, i.e., $q \in \mathcal{V}_S$ or $n \in \mathcal{V}_S$: this means an exact value for $s(q, n)$ is already precomputed



(a) Without approximation: involves many repetitive computations



(b) With approximation: involves retrieving approximate similarities of vertex separators to nodes

Figure 5.2: Approximate RoleSim* node-pair similarity retrieval by caching the similarities between vertex separators to all nodes and using triangle inequality to compute approximate results for node-pairs $(2, 3)$ in case 1, $(2, 1)$ in case 2, and $(2, 4)$ in case 3

2. Both nodes are in the same partition, say $q \in \mathcal{G}_1$ and $n \in \mathcal{G}_1$: this means an approximate value for $s(q, n)$ can be computed considering $\mathcal{G}_1$ as the input graph and discarding all nodes/edges from $\mathcal{G}_2$

3. Both nodes are in different partitions, say $q \in \mathcal{G}_1$ and $n \in \mathcal{G}_2$: this means an approximate (lower bound) value for $s(q, n)$ can be directly computed from the precomputed exact values of $s(v, *)$ by making use of the triangle inequality property of RoleSim*

Figure 5.2 depicts how numerous additional similarity computations are avoided by retrieving the values stored for nodes in $\mathcal{V}_S$ and using these to compute approximate results. In cases 1 and 2 when $\mathcal{G}$ is pruned into $\mathcal{G}_1$, the nodes and edge connections that are discarded can no longer contribute to the approximate similarity computation during the RoleSim* traversals on the partitioned graph, and only the precomputed values at the vertex separator may be used. The case 3, considering $q \in \mathcal{G}_1$ and $n \in \mathcal{G}_2$ and precomputed exact values of $s(v, *)$, is explained in further detail:

Using triangle inequality from Equation 5.5, $\forall v \in \mathcal{V}_S$ the following result holds:

$$s(q, n) \geq s(q, v) + s(v, n) - 1$$

Hence, using the precomputed values of $s(v, *)$, the score for $s(q, n)$ must have a lower bound as the largest of these values, or:

$$s(q, n) \geq \max_{v \in V_S} \big(s(q, v) + s(v, n) - 1\big)$$

This can be extended to obtain approximate results using any p-way partitioning. The vertex separator set is added (along with its cached precomputed similarity scores to all nodes) into

each of the subgraphs after using a p-way partitioning scheme, and the same procedure is followed for pruning.

Exact single-source similarity search

The challenge of returning an exact solution for the most similar nodes to a query q can also be considered. The vertex separator set $\mathcal{V}_S$ is taken as a set of keys whose similarities to all nodes are precomputed. From the triangle inequality:

$$d(q,n) \geq |d(v,n) - d(q,v)| \tag{5.6}$$

Hence, a lower bound may be obtained on $d(q,n)$, by computing:

$$d(q,n) \geq \max_{v \in V_S} |d(v,n) - d(q,v)| \tag{5.7}$$

Given some $s_{\min}$, the goal is to find all nodes n_i such that $s(q,n_i) \geq s_{\min}$. That is, $d(q,n_i) \leq \xi$ where $\xi = 1 - s_{\min}$ is the distance threshold for pruning. The lower bounds $d(q,n_i)$ for all nodes n_i are given by Equation 5.7. Any lower bound greater than ξ allows the node n_i to be pruned from the candidate list. This resulting candidate set is denoted as $\mathcal{V}_k$. In large graphs, with careful partitioning to select a small number of vertex separators ($|\mathcal{V}_k| \ll |\mathcal{V}|$), the number of distance computations is substantially reduced.

This approach can be applied to prune partitions while identifying the top-k similar nodes to query q. Consider subgraphs $\mathcal{G}_j$ obtained after partitioning $\mathcal{G}$, each containing a single vertex separator v. Using precomputed values for $d(v,*)$, Equation 5.6 gives the lower bounds of $d(q,n_i)$ for $n_i \in \mathcal{V}_{\mathcal{G}_j}$ ($j = 1, \cdots, p$ respectively for each of the p partitions). The minimum of all these distances within a partition gives a lower bound value that helps to index and order the partitions:

$$d(q,n_i) \geq \min_{n_i \in \mathcal{V}_{\mathcal{G}_j}} |d(v,n_i) - d(q,v)|$$

These lower bounds may be denoted as $\xi_{\mathcal{G}_j}$ for each subgraph $\mathcal{G}_j$. Without loss of generality, suppose $\xi_{\mathcal{G}_1} > \xi_{\mathcal{G}_2}$ for a 2-way partitioning of $\mathcal{G}$. Thus, every node in $\mathcal{G}_1$ is necessarily at least $\xi_{\mathcal{G}_1}$ distance away from q. First, the exact distances are computed for all nodes in $\mathcal{G}_2$. If k such nodes all have a distance to q that is smaller than $\xi_{\mathcal{G}_1}$, then nodes of $\mathcal{G}_1$ need not be considered, and the resulting top-k can be directly returned. If not, that is if any nodes of $\mathcal{G}_2$ are at a distance higher than the lower bound of the next partition (here, $\mathcal{G}_1$), then nodes of $\mathcal{G}_1$ must be processed. These nodes are then inserted into the top-k ranking based on the computed distances. A similar process continues through the ordered set of $\xi_{\mathcal{G}_j}$ values for p-way partitioned data.

5.3 Technique for evaluation of solution quality

A brief departure is needed from discussions of the partitioning approach to consider the impact that sampling can have on the graph-based task, in this case SSRS*. It is important

to study how sampling strategies (hypergraph-based or otherwise) can affect the results obtained for a given task, before such sampling is performed indiscriminately to prune graph structures. For this to be possible, the quality of results obtained from different similarity measures needs to be quantified. A novel unsupervised evaluation setting is devised using self-similarity as the ground truth. This allows a study into the effect of sampling the immediate neighbourhood of a query node on similarity scores in RoleSim*, compared with SimRank and RoleSim.

This evaluation is inspired by the tactic of determining duplicate nodes in a network simply by examining their neighbourhoods for similar patterns. In many applications, the underlying network contains duplicate entities with noisy, incomplete, and partially overlapping information, such as in a social network that has been scraped from multiple sources. The similarity between duplicates is expected to be high, and can help to identify them as such. Here, duplicate entities are considered as separate nodes, where each duplicate has some sampling of the total set of neighbouring edges available to the node. For example, in a co-purchasing product graph (`AMZ`), duplicates may exist when merging multiple e-commerce sources, or when identical products are sold by different sellers. This indicates that each of these duplicate products were frequently purchased along with certain other products as they share some common neighbours. As another example, differently formatted author names or multiple sources for a paper can lead to duplicates in co-authorship and citation networks (`DBLP`).

Consider a single query node q. A duplicate node q' is created and added to the graph. This q' is connected to some proportion of the total number of neighbours of q. The proportion η is the sampling ratio parameter. Henceforth, q' is referred to as the 'sampled clone'. The similarity scores of q to all other points in the graph can now be computed using SimRank, RoleSim, RoleSim*, or any other chosen measure. In particular, the proposed evaluation scheme is interested in whether the similarity between q and its sampled clone q' can be preserved and identified by the chosen measure, while varying the sampling ratio η.

5.4 Experimental evaluation

The objective is to quantify the effectiveness of the similarity measures using different partitioning strategies, and to evaluate the tradeoff between the quality of approximate results and the improvement achieved in running time, when the input graph is partitioned or pruned in different ways.

Datasets

Two real-world datasets are used in the evaluation. The first is **`DBLP`**, an undirected collaboration graph taken from DBLP bibliography.[1] A co-authorship subgraph is extracted from six top conferences in computer science for the time period of 2018–2020. An edge connected two authors (nodes) if they have co-authored a paper. The second dataset is **`AMZ`**, a directed

[1] https://dblp.uni-trier.de/db/

co-purchasing graph derived from Amazon.[2] Each node represents a product, with an edge connecting $i \rightarrow j$ if product j appears in the frequent co-purchasing list of product i.

Table 5.1: Dataset properties

Dataset	#Nodes	#Edges	Type
DBLP	2,372	7,106	Undirected
AMZ	5,086	8,970	Directed

Baselines

The similarity scores obtained using RoleSim* (**RS***) are compared with those from SimRank (**SR**) and RoleSim (**RS**). These methods compute respective similarities as described earlier. In particular, the single-source RoleSim* method (**SSRS***) is also examined in different settings.

METIS [84] and rpPaToH [141] are used as the graph and hypergraph partitioning tools respectively, using their vertex separator or replication functionality.

Setup

All experiments are conducted on an Ubuntu 20.04 machine with 8 Intel 1.70 GHz CPUs and 8GB RAM.

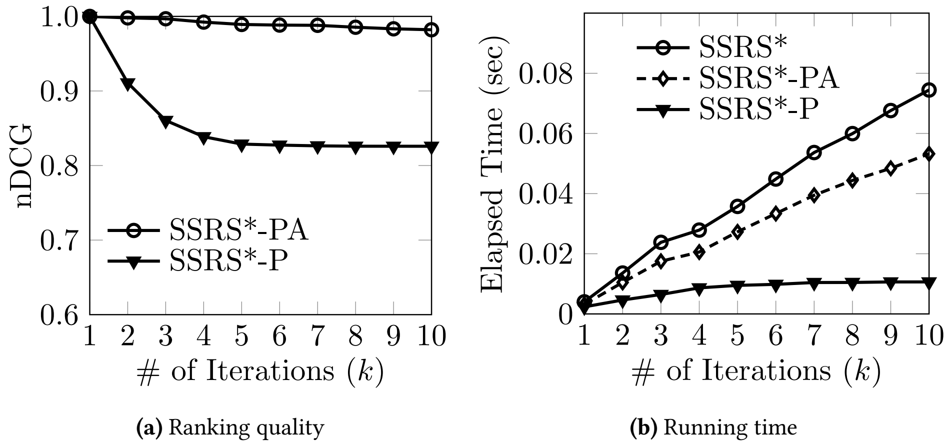
The similarity results are aggregated over a query workload on DBLP and AMZ graphs respectively, where query nodes are chosen at random from the graph. Each experiment is repeated 5 times and the average is reported.

Performance of SSRS* with partitioning

A partitioning scheme (using the triangle inequality property) is applied for approximate SSRS* computation. First, a graph-based partitioning method is tested. METIS is used to generate 2-way graph partitioning of the graph using the vertex separator method. Similarity scores are precomputed from these vertex separator nodes to all nodes, and these cached values are retrieved during single-source RS* computations. Second, a hypergraph-based partitioning method is tested in a similar fashion. For this, rpPaToH is used to generate 2-way hypergraph partitioning with vertex replication. These replicated vertices are treated in exactly the same manner as the vertex separators described above, to precompute and cache similarity scores.

For both partitioners, the resulting partitioning is made to produce 10 separator (replicated) vertices. It should be noted that the number of separator vertices has an impact on the quality of results. The 2-way partitioning of the given data using METIS results in 10 vertex separators. The replication ratio constraint in rpPaToH is used to tweak the number of resulting replicated vertices obtained from the hypergraph model, and made to match the number of vertex separators generated by METIS for the graph model. If a greater number of vertices were replicated (and therefore had precomputed scores cached), this would lower the running time while also negatively affecting the quality of the approximate solutions. The running time decreases due to the easier retrieval of cached values with fewer exact

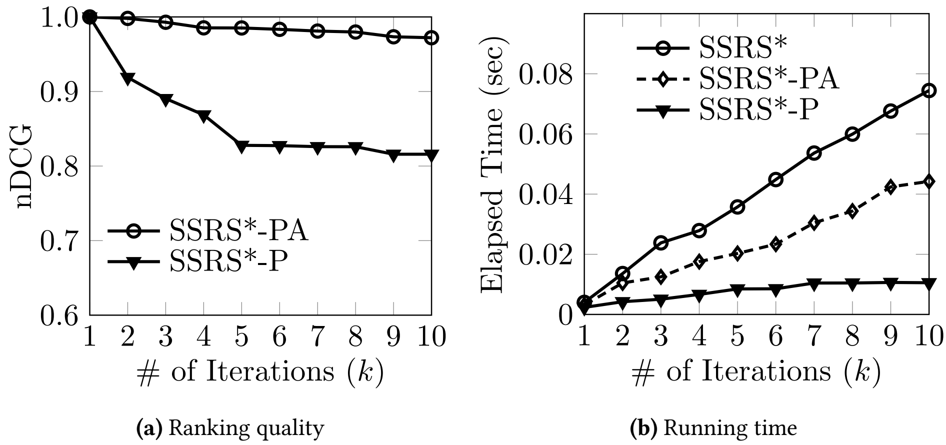
[2] https://www.amazon.co.uk/



(a) Ranking quality

(b) Running time

Figure 5.3: Performance comparison with graph-based partitioning on DBLP



(a) Ranking quality

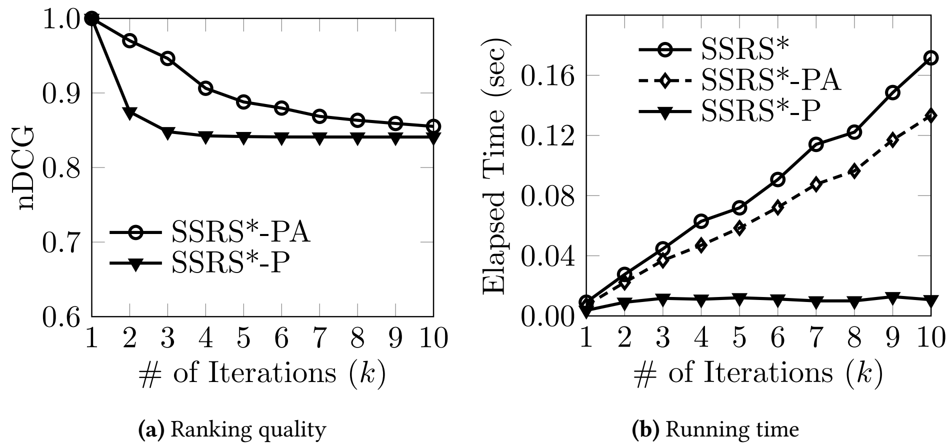
(b) Running time

Figure 5.4: Performance comparison with hypergraph-based partitioning on DBLP

computations needing to be performed. At the same time, avoiding exact computations causes the quality of the resulting similarity scores to drop.

The average ranking quality based on normalised discounted cumulative gain (nDCG) is commonly used to evaluate whether similarity measures are effective and robust [189]. The accuracy (nDCG) and efficiency (running time) of SSRS* over real datasets DBLP and AMZ are evaluated by randomly sampling 100 queries from each dataset. With varying number of iterations k, single-source RoleSim* similarities $\{s_k(*, q)\}$ are computed with respect to each query q. The SSRS*-P and SSRS*-PA methods are then used to test the performance on the partitioned graphs. Choosing these SSRS* similarities $\{s_k(*, q)\}$ as the baseline, the average value of nDCG is evaluated for both SSRS*-P and SSRS*-PA over 100 queries on each dataset. SSRS*-P denotes an approach where only the pruned subgraph is considered for similarity computation (strict data locality without inter-partition communication), while SSRS*-PA denotes an approach where access to neighbouring nodes in other partitions is allowed during the similarity computation (inter-partition access is permitted).

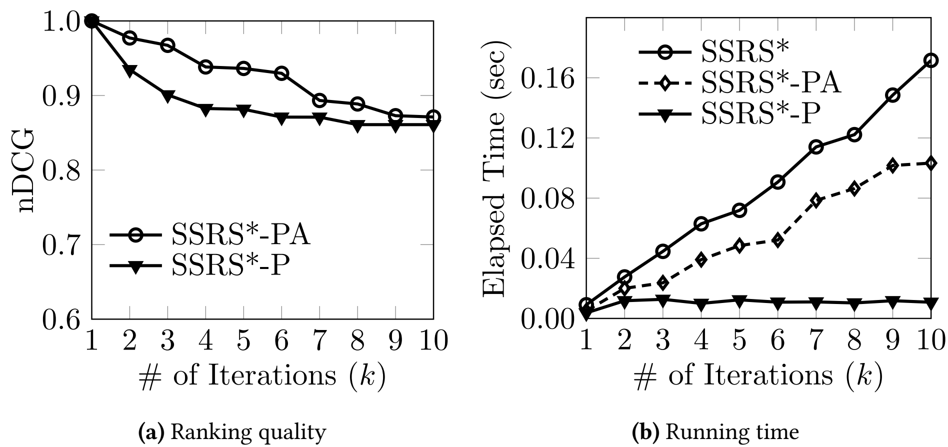
Considering the graph-based partitioning approach, Figures 5.3a and 5.5a indicate that, on these datasets, SSRS*-P achieves close to 85% accuracy in terms of nDCG for top-100 results.



(a) Ranking quality

(b) Running time

Figure 5.5: Performance comparison with graph-based partitioning on AMZ



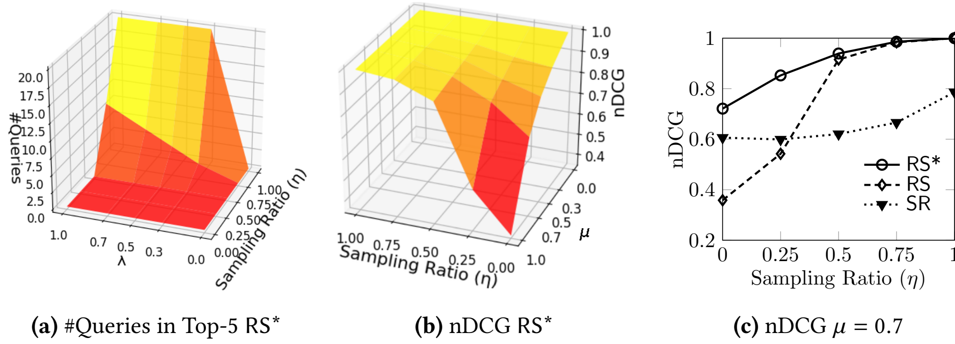
(a) Ranking quality

(b) Running time

Figure 5.6: Performance comparison with hypergraph-based partitioning on AMZ

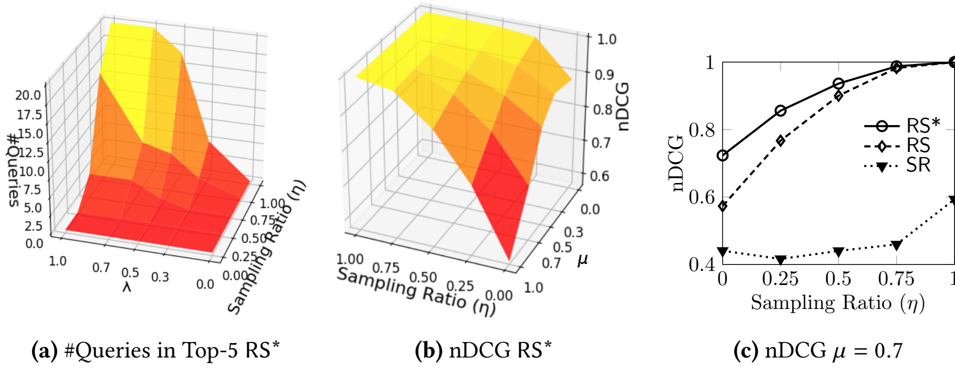
SSRS*-PA is more accurate, however it incurs much higher computational time due to more edge connections being taken into consideration. As seen from Figures 5.3b and 5.5b, a partitioning approach like SSRS*-P may offer a more scalable computation of approximate similarity scores even for large number of iterations.

Figures 5.4a and 5.6a show that accuracy remains largely the same for the hypergraph-based partitioning method as with the graph-based method. For the hypergraph partitioner, efficiency is improved over the graph partitioner with SSRS*-PA on both datasets (by 10–15%) as seen in Figures 5.4b and 5.6b. Hypergraph-based partitioners have the benefit of encoding communications more accurately, thus the selection of the replicated vertices is likely to better reflect their importance within the neighbourhood access workload pattern. The improved efficiency is more stark in the case of the directed graph AMZ, even though both METIS and rpPaToH treat the graph/hypergraph as undirected. Therefore, this is more reasonably attributed to their relative sparsity patterns, as the AMZ graph has sparser connections than DBLP. Hypergraph-based partitioners are known to be especially powerful for sparse adjacency matrices as they can represent the exact communication volume.



(a) #Queries in Top-5 RS* **(b)** nDCG RS* **(c)** nDCG $\mu = 0.7$

Figure 5.7: Effect of sampling ratio η and weight μ on ranking quality on DBLP



(a) #Queries in Top-5 RS* **(b)** nDCG RS* **(c)** nDCG $\mu = 0.7$

Figure 5.8: Effect of sampling ratio η and weight μ on ranking quality on AMZ

Performance of SSRS* with sampled neighbourhood

The solution quality of different similarity measures can be evaluated using the previously described novel unsupervised setting. Here, the number of queries where the sampled clone q' is deemed similar to query q intuitively studies how much structural information about the query node is successfully ascertained by the similarity model. For example, an indicator of good semantic accuracy of the similarity model could be when a sampled clone q' is present in the top-k similar nodes list of q, or when the amount of overlap between the top-k lists of q and q' is high. First, η is varied for q' with a step size of 0.25 (and ensuring no orphaned nodes). This results in some overlap of neighbourhoods as the value of η grows towards 1. The values $\mu = \{0.0, 0.3, 0.5, 0.7, 1.0\}$ are considered for the relative weight balancing parameter in the two-part calculation of RS*.

Figure 5.7a presents the number of queries out of the total workload of 20 queries, on the undirected DBLP graph, where the sampled clone q' is among the top-5 similarity scores for the query q. This top-5 list is based on all nodes in the graph, i.e., over 2K for DBLP. Similarly, Figure 5.8a shows results on the directed AMZ graph. Both sets of plots depict that with higher η, there are more such queries. However, the greater overlap in neighbourhood is not the only contributing factor. RS* is heavily impacted when there is a large mismatch in the sampled neighbourhood sizes (as is RS). This is confirmed through an exploratory study where results are obtained on varying the sampling of both q and q' together (sampling η and η' neighbouring edges respectively, which both can now vary from 0 to 1). Despite random samples of neighbourhoods, the results peak only when the neighbourhood sizes

are close to each other (i.e., η and η' are equal). More specifically, the exact nature of the neighbouring nodes themselves appears less important compared to the relative structure of connectivity patterns within the neighbourhood.

Next, the impact of sampling ratio η and weight balancing parameter μ on the ranking quality of RoleSim* are tested. The nDCG is plotted, considering top-100 similar nodes of the sampled clone and comparing this to the baseline original query. The trend (with respect to η) seen in Figure 5.7b and Figure 5.8b for $\mu = 1$ is observed to resemble that obtained for RoleSim, and the trend for $\mu = 0.5$ is close to that obtained for SimRank.

A fixed value of $\mu = 0.7$ is also considered, which demonstrates that RS* has higher ranking quality compared to SR and RS, with respect to the average nDCG. Figure 5.7c with undirected `DBLP` graph shows that RS* produces a more consistent nDCG even with small η. For the directed `AMZ` graph in Figure 5.8c too, RS* remains more stable while RS shows significant changes with sampling ratio, and the performance of SR is poor throughout. The results together indicate that for the challenge of identifying duplicate entities, RoleSim* is best suited to correctly identify a match. When presented with a noisy sample of edge connections from the duplicate node, even a small sample of edges from the duplicate can produce good matching results. That is, regardless of the node itself, the topology of the neighbourhood is important for RoleSim*, and this should be preserved by the partitioning scheme employed to scale up RoleSim* across distributed subgraphs. As previously determined, the partitioning policy of allowing/disallowing communications across these subgraphs is also important to tradeoff efficiency and accuracy of the approximate results.

5.5 Discussion

In this chapter, partitioning approaches are applied on static graphs and evaluated using a graph analytics similarity search workload. The goal is to examine the tradeoff between solution quality and running time of the approximate single-source RoleSim* similarity search algorithm.

Both graph-based and hypergraph-based partitioners are employed, with the set of replicated vertices being used to cache precomputed similarity scores. It is demonstrated on two real-world datasets that the SSRS* algorithm on a pruned subgraph is considerably faster than the baseline version run on the complete graph. However, observing strict data locality (SSRS*-P) hurts the accuracy performance (nDCG) and lowers the solution quality of the approximate results compared to the exact results by 15-20%. Allowing some inter-partition communication (SSRS*-PA) during the similarity computations improves the solution quality. This comes at the cost of efficiency, as the communication overheads and extra neighbourhood accesses incur longer running time.

Partitioning the search graph is thus shown to be a useful strategy empirically, and this is also evident from the nature of the sparse matrix computations required for SimRank, RoleSim, and RoleSim*. Hypergraph-based partitioning is especially promising, due to the exact communication volumes being encoded by this method when sparse matrix multiplication is to be performed. The hypergraph partitioner is found to slightly lower the running time of SSRS*-PA, which can be explained by its ability to choose better vertices for caching.

The findings in this chapter pave the way for further research into novel hypergraph-based partitioning and sampling optimisations on static graphs. One such method is presented in the following chapter specifically targeting GNN workloads.

In addition to these results, this chapter also provides a brief exploration of the effect that neighbourhood sampling can have on graph-based tasks. An unsupervised evaluation method is proposed to quantify how well a similarity measure is able to detect duplicate nodes. These duplicates are sampled clones containing a subset of the neighbourhood of the original nodes. The behaviour of the search algorithm suggests that relative structure of connectivity patterns is an important indicator of similarity, and even a small sample of edge connections can be exploited to derive accurate node matches. The topology of the neighbourhood, rather than the node itself, is important for RoleSim*. Therefore, the partitioning scheme on such similarity measures should preserve this topology information where possible to ensure quality of approximate results is maintained even on distributing the computations. Different communication policies across these partitions additionally cause a tradeoff between efficiency and accuracy. This study is therefore beneficial to understand where sampling and data locality methods can be safely enforced in future chapters.

Chapter 6

Hypergraph-based Partitioning on Static Graphs for GNNs

> *We are, at this moment, both creating and solving problems* ***faster*** *than we ever have before. So your job – the only thing anyone can ask of you as a human – is to solve more problems than you create.*
>
> Hank Green, *vlogbrothers (Hello Future Dead Person)*

The previous chapters dealt with graph analytics workloads, with explorations of hypergraph-based partitioning and sampling optimisations for both static and dynamic graph data. In this chapter, the focus is on graph learning applications. A hypergraph-based strategy is developed for communication-efficient distributed GNN training on static graphs. Here, an asynchronous communication scheme is designed using hypergraph-based partitioning to account for the unique access patterns in GNNs that differ from those of traditional analytics tasks on which graph/hypergraph partitioners are commonly tested.

The scale of large graphs, including their multi-dimensional features for the vertices, necessitates the use of distributed-memory systems for successfully training graph-based learning models [112, 194]. GCNs are particularly popular for deep learning on graphs, and have become the de facto standard for learning graph representations. During the feedforward and backpropagation phases in GCN training, the graph convolution operation involves message passing and aggregation steps that induce irregular data accesses due to complex graph interconnectivity. Existing systems use graph partitioning algorithms designed for traditional graph algorithm workloads (e.g., connected components or shortest paths), that do not take the unique challenges of complex GCN data access patterns into consideration. Therefore, intelligent partitioning and message passing strategies need to be employed, that improve data locality and avoid broadcasts, to achieve a communication-efficient distributed-memory parallel inference and training solution.

To address this, a highly parallel GCN training algorithm is introduced that scales to large processor counts. In the proposed solution, the large adjacency and vertex-feature matrices are partitioned among processors. The edge-cut partitioning of the graph is exploited to make use of asynchronous non-blocking point-to-point communication operations between processors for better scalability. To further minimise the parallelisation overheads, a sparse matrix partitioning scheme is introduced based on a hypergraph partitioning model for full-batch training. The merits of such a hypergraph model are shown over the standard graph partitioning model which does not accurately encode the communication costs of transmitting features across processors.

In order to capture the randomness when communication operations are performed for mini-batch training instead of full-batch training, this chapter also includes a stochastic hypergraph model. This model is based on a hypergraph sampling and construction technique that can be utilised for any mini-batch sampling strategy. It encodes the *expected* communication volume for additional optimisations in parallel mini-batch training.

The solutions in this work focus on large-scale CPU clusters, commonly used for big sparse problem instances in scientific computing, since research towards adapting existing, relatively inexpensive supercomputing systems towards deep learning is gaining attention. Nevertheless, the solutions are also demonstrated on GPU clusters. The full-batch and mini-batch solutions significantly improve efficiency and scalability over the state-of-the-art algorithms. The communication scheme devised here is easily applicable on any message passing GNN models operating on a variety of large-scale real-world graph data, such as citation networks, social networks, and road networks.

6.1 Preliminaries

GCN feedforward phase

GCNs generalise the convolution operation to graphs having arbitrary size and topology. An adjacency matrix can be used to describe the (sparse) edge connections along which data aggregation takes place for every layer in the neural network. Let $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ denote the adjacency matrix of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ which consists of $|\mathcal{V}| = n$ vertices. Vertex set $\mathcal{V}$ is associated with a feature matrix $\boldsymbol{H}^\ell \in \mathbb{R}^{n \times d_\ell}$ for every GCN layer ℓ, rows of which correspond to d_ℓ-dimensional vertex features.

In a simple GCN [89], given an input feature matrix $\boldsymbol{H}^0$, feedforward of GCN is defined for layers $\ell = 1, 2, \ldots L$ as

$$\begin{aligned} \boldsymbol{Z}^\ell &= \widehat{\boldsymbol{A}} \boldsymbol{H}^{\ell-1} \boldsymbol{W}^\ell \\ \boldsymbol{H}^\ell &= \rho(\boldsymbol{Z}^\ell) \end{aligned} \quad (6.1)$$

Some re-normalisation tricks (to deal with exploding/vanishing gradient problems) and self-loops (for computations at a node to consider the data at the node itself) are introduced into A. Matrix $\widehat{\boldsymbol{A}}$ is formed as $\widehat{\boldsymbol{A}} = \boldsymbol{D}^{-\frac{1}{2}} \widetilde{\boldsymbol{A}} \boldsymbol{D}^{-\frac{1}{2}}$ for normalisation, where matrix $\widetilde{\boldsymbol{A}} = \boldsymbol{A} + \boldsymbol{I}$ corresponds to the adjacency matrix with self loops and matrix $\boldsymbol{D}(i, i) = \sum_j \widetilde{\boldsymbol{A}}(i, j)$ corresponds to the

diagonal matrix of vertex degrees. To ease the notation, $\boldsymbol{A}$ is used instead of $\widehat{\boldsymbol{A}}$ to denote the normalised adjacency matrix.

In Equation 6.1, only the $\boldsymbol{A}$ matrix is sparse and the remaining matrices are dense. SpMM $\boldsymbol{A}\boldsymbol{H}^{\ell-1}$ combines feature vectors for each vertex (itself and neighbours). The resulting combined features are then involved in a DMM and multiplied by trainable parameter matrix $\boldsymbol{W}^{\ell} \in \mathbb{R}^{d_{\ell-1} \times d_{\ell}}$. Finally, a non-linear activation function $\rho(\cdot)$ is applied to each element of matrix $\boldsymbol{Z}^{\ell}$ to compute $\boldsymbol{H}^{\ell}$.

GCN backpropagation phase

The backpropagation phase requires a gradient matrix $\boldsymbol{G}^{L} \in \mathbb{R}^{n \times d_L}$ which is computed as

$$\boldsymbol{G}^{L} = \nabla_{\boldsymbol{H}^L}\mathbf{J} \odot \rho'(\boldsymbol{Z}^{L})$$

where $\nabla_{\boldsymbol{H}^L}\mathbf{J}$ denotes the matrix of derivatives of the loss function $\mathbf{J}$ with respect to output features in $\boldsymbol{H}^{L}$, $\rho'(\cdot)$ denotes the derivative of the activation function, and symbol $\odot$ denotes element-wise multiplication (i.e., Hadamard product). Gradient matrices for the preceding layers for $\ell = L, L-1, \ldots, 1$ are recursively computed as

$$\begin{aligned} \boldsymbol{S}^{\ell} &= \boldsymbol{A}\boldsymbol{G}^{\ell}(\boldsymbol{W}^{\ell})^{T} \\ \boldsymbol{G}^{\ell-1} &= \boldsymbol{S}^{\ell} \odot \rho'(\boldsymbol{Z}^{\ell-1}) \end{aligned} \tag{6.2}$$

In Equation 6.2, SpMM is performed with matrices $\boldsymbol{A}$ and $\boldsymbol{G}^{\ell}$, and the resulting matrix is used in DMM with $(\boldsymbol{W}^{\ell})^{T}$. Each gradient matrix $\boldsymbol{G}^{\ell} \in \mathbb{R}^{n \times d_{\ell}}$ is used to update parameter matrix $\boldsymbol{W}^{\ell}$ by the following set of gradient update rules

$$\begin{aligned} \Delta\boldsymbol{W}^{\ell} &= (\boldsymbol{H}^{\ell-1})^{T}\boldsymbol{A}\boldsymbol{G}^{\ell} \\ \boldsymbol{W}^{\ell} &\leftarrow \boldsymbol{W}^{\ell} - \alpha\Delta\boldsymbol{W}^{\ell} \end{aligned}$$

where $\Delta\mathbf{W}^{\ell}$ denotes the matrix of derivatives of the loss function $\mathbf{J}$ with respect to parameters in matrix $\boldsymbol{W}^{\ell}$, and α denotes the learning rate. It is important to note that, if the input graph is directed, transpose $\boldsymbol{A}^{T}$ is used instead of $\boldsymbol{A}$ in backpropagation (a more detailed description can be found in [154]).

6.2 Data-parallel GCN training algorithm

A highly parallel algorithm is introduced for training GCNs on distributed-memory systems. In this section, the feedforward and backpropagation steps of the proposed parallel GCN training algorithm are presented. The solution achieves scalability by replacing the blocking broadcast communications in existing approaches with non-blocking point-to-point communications for parallel SpMM and transferring only the necessary data with minimal number of messages between processors. The solution employs a one-dimensional (1D) partitioning on large adjacency, vertex-feature, and gradient matrices for parallel SpMM computations in feedforward and backpropagation phases. It replicates parameter matrices across processors

due to their relatively smaller sizes. This enables data locality for performing DMM computations without any communication. Allreduce communication is needed for aggregating gradients, which has a negligible cost compared to the communication costs incurred in parallel SpMM.

GCN parallel feedforward

The proposed parallel feedforward algorithm executes on p processors each of which is denoted by P_m for $m = 1, 2, \dots, p$. Adjacency matrix $\boldsymbol{A}$ and vertex feature matrices $\boldsymbol{H}^\ell$ for all layers $\ell = 0, 1, \dots, L$ are 1D row-wise partitioned among processors where each processor P_m stores submatrices $\boldsymbol{A}_m \in \mathbb{R}^{n\times n}$ and $\boldsymbol{H}_m^\ell \in \mathbb{R}^{n\times d_\ell}$, which only contain subsets of rows of matrices $\boldsymbol{A}$ and $\boldsymbol{H}^\ell$. Adjacency matrix and feature matrices are conformably partitioned so that if row $\boldsymbol{A}(i,:)$ is assigned to submatrix $\boldsymbol{A}_m$, then the corresponding feature vectors $\boldsymbol{H}^\ell(i,:)$ for all layers ℓ are assigned to submatrices $\boldsymbol{H}_m^\ell$, respectively (i.e., $\boldsymbol{A}(i,:) \in \boldsymbol{A}_m \Leftrightarrow \boldsymbol{H}^\ell(i,:) \in \boldsymbol{H}_m^\ell\ \forall \ell$). Parameter matrices $\boldsymbol{W}^\ell$ for all layers ℓ are replicated and stored by all processors due to their relatively smaller sizes.

The matrix partitioning scheme encodes a vertex-partitioning on graph $\mathcal{G}$, since the rows $\boldsymbol{A}(i,:)$ and $\boldsymbol{H}^\ell(i,:)$ denote the adjacency list and features of vertex $v_i \in \mathcal{V}$. Moreover, this partitioning also induces a task partitioning in the feedforward phase: If a vertex v_i is assigned to a processor P_m, the task of computing row $\boldsymbol{Z}(i,:)^\ell$ of intermediate matrix $\boldsymbol{Z}^\ell$ in layer ℓ is performed by processor P_m where row $\boldsymbol{Z}(i,:)^\ell$ is computed as

$$\boldsymbol{Z}^\ell(i,:) = \left(\sum_{j \in \mathrm{cols}(\boldsymbol{A}(i,:))} \boldsymbol{A}(i,j)\boldsymbol{H}^{\ell-1}(j,:) \right) \boldsymbol{W}^\ell$$

Hence, to compute submatrix $\boldsymbol{Z}_m^\ell$, processor P_m needs to receive all $\boldsymbol{H}^{\ell-1}$-matrix rows corresponding to all nonzero column indices in $\boldsymbol{A}_m$, which are not locally stored in $\boldsymbol{H}_m^{\ell-1}$. Let $\boldsymbol{H}_{nm}^{\ell-1} \in \mathbb{R}^{n\times d_{\ell-1}}$ denote the submatrix consisting of rows that are needed to be transferred from processor P_n to P_m. That is, submatrix $\boldsymbol{H}_{nm}^{\ell-1}$ contains subset of rows of $\boldsymbol{H}_n^{\ell-1}$ corresponding to the intersection of nonzero row indices of $\boldsymbol{H}_n^{\ell-1}$ and column indices of $\boldsymbol{A}_m$. More formally, $\exists\, i \in \mathrm{rows}(\boldsymbol{H}_{nm}^{\ell-1})$ if $i \in \mathrm{cols}(\boldsymbol{A}_m) \cap \mathrm{rows}(\boldsymbol{A}_n)$. Non-blocking point-to-point communications are employed to transfer these submatrices between processors. After processor P_m receives submatrix $\boldsymbol{H}_{nm}^{\ell-1}$ from each processor P_n for all $n \neq m$ such that $\boldsymbol{H}_{nm}^{\ell-1} \neq \boldsymbol{0}$, P_m performs multiplication

$$\boldsymbol{Z}_m^\ell = (\boldsymbol{A}_m \boldsymbol{H}_m^{\ell-1} + \sum_{n \neq m} \boldsymbol{A}_m \boldsymbol{H}_{nm}^{\ell-1}) \boldsymbol{W}^\ell$$

to compute submatrix $\boldsymbol{Z}_m^\ell \in \mathbb{R}^{n\times d_\ell}$. Then, P_m applies the nonlinear activation function $\boldsymbol{H}_m^\ell = \rho(\boldsymbol{Z}_m^\ell)$ to proceed to the next layer.

To manage sparse point-to-point communication operations, each processor P_m is provided with sets $\mathcal{S}_m$ and $\mathcal{R}_m$ which are computed before training with respect to the partitioning of adjacency matrix $\boldsymbol{A}$ among processors. Set $\mathcal{S}_m$ is composed of diagonal matrices $\boldsymbol{X}_{mn} \in \mathbb{R}^{n\times n}$ for each processor $P_n \neq P_m$. Matrix $\boldsymbol{X}_{mn}$ is used in a special matrix multiplication to

determine which local $\boldsymbol{H}_m^{\ell-1}$-rows to be sent by processor P_m to P_n. Formally,

$$\mathcal{S}_m = \{\boldsymbol{X}_{mn} \mid \boldsymbol{X}_{mn} \neq \mathbf{0} \wedge \boldsymbol{X}_{mn}(i,i) = 1 \\ \forall i \in \text{cols}(\boldsymbol{A}_n) \cap \text{rows}(\boldsymbol{A}_m)\}$$

That is, the ith diagonal entry $\boldsymbol{X}_{mn}(i,i)=1$ if the intersection of nonzero row and column indices of matrices $\boldsymbol{A}_m$ and $\boldsymbol{A}_n$ contains index i, otherwise it is set to zero. Set $\mathcal{R}_m$ is composed of processors from which P_m receives at least one message. Formally,

$$\mathcal{R}_m = \{P_n \mid P_n \neq P_m \wedge cols(\boldsymbol{A}_m) \cap rows(\boldsymbol{A}_n) \neq \emptyset\}$$

That is, processor P_n is included in $\mathcal{R}_m$ if the intersection of nonzero row and column indices of matrices $\boldsymbol{A}_m$ and $\boldsymbol{A}_n$ is nonempty, and processor P_m receives at least one row of $\boldsymbol{H}_n^{\ell-1}$ from processor P_n.

Algorithm 6.1 describes the proposed parallel feedforward algorithm. The GraphBLAS [36] library is used to perform the sparse matrix operations. In lines 3–5, to overlap communication by computation, a non-blocking communication is performed for each diagonal matrix $\boldsymbol{X}_{mn} \in \mathcal{S}_m$ by processor P_m to send required $\boldsymbol{H}_m^{\ell-1}$-matrix rows to processor P_n. Matrix $\boldsymbol{H}_{mn}^{\ell-1}$ is formed through a specialised matrix multiplication $\boldsymbol{H}_{mn}^{\ell-1} = \boldsymbol{X}_{mn} \otimes \boldsymbol{H}_m^{\ell-1}$. By this matrix multiplication, if the ith diagonal entry is $\boldsymbol{X}_{mn}(i,i)=1$, then the ith row $\boldsymbol{H}_m^{\ell-1}$ is copied into matrix $\boldsymbol{H}_{mn}^{\ell-1}$. Operator $\otimes$ denotes that the matrix multiplication is performed under a semiring defined by GraphBLAS, to replace the multiplication operator with a copy operator that will directly carry the second operand to the resulting variable without multiplying (i.e., $z = x \times y \Rightarrow z = y$). In line 6, local matrix multiplication $\boldsymbol{Z}_m^{\ell} = \boldsymbol{A}_m \boldsymbol{H}_m^{\ell-1} \boldsymbol{W}^{\ell}$ is performed without waiting for the non-blocking communication operations to complete. Matrix $\boldsymbol{Z}_m^{\ell}$ is incomplete at this stage and its computation is finalised after receiving all necessary data. In lines 7–9, processor P_m receives $\boldsymbol{H}_{nm}^{\ell-1}$ from each $P_n \in \mathcal{R}_m$, and performs multiplication and addition $\boldsymbol{Z}_m^{\ell} = \boldsymbol{Z}_m^{\ell} + \boldsymbol{A}_m \boldsymbol{H}_{nm}^{\ell-1} \boldsymbol{W}^{\ell}$ to compute the final matrix $\boldsymbol{Z}_m^{\ell}$. The activation function ρ is then applied in line 10 to generate $\boldsymbol{H}_m^{\ell}$ for layer ℓ.

Algorithm 6.1: GCN Parallel Feedforward

```
1  forall processors P_m in parallel do
2  |  for ℓ = 1 to L do
3  |  |  foreach X_mn ∈ S_m do
4  |  |  |  H_mn^(ℓ-1) = X_mn ⊗ H_m^(ℓ-1)
5  |  |  |  Non-blocking send H_mn^(ℓ-1) to processor P_n
6  |  |  Z_m^ℓ = A_m H_m^(ℓ-1) W^ℓ
7  |  |  foreach P_n ∈ R_m do
8  |  |  |  Receive H_nm^(ℓ-1) from processor P_n
9  |  |  |  Z_m^ℓ = Z_m^ℓ + A_m H_nm^(ℓ-1) W^ℓ
10 |  |  H_m^ℓ = ρ(Z_m^ℓ)
```

GCN parallel backpropagation

In the backpropagation phase, similar to the vertex feature matrices, gradient matrices $\boldsymbol{G}^{\ell}$ for each layer ℓ are row-wise partitioned among processors where each processor P_m holds

submatrix $\boldsymbol{G}_m^\ell \in \mathbb{R}^{n \times d_\ell}$ in each layer ℓ. Gradient matrix $\boldsymbol{G}^\ell$ and adjacency matrix $\boldsymbol{A}$ are conformably partitioned so that if row $\boldsymbol{A}(i,:)$ is assigned to submatrix $\boldsymbol{A}_m$, then row $\boldsymbol{G}^\ell(i,:)$ is assigned to submatrix $\boldsymbol{G}_m^\ell$ (i.e., $\boldsymbol{A}(i,:) \in \boldsymbol{A}_m \Leftrightarrow \boldsymbol{G}^\ell(i,:) \in \boldsymbol{G}_m^\ell\ \forall \ell$). Hence, the task of computing row $\boldsymbol{S}(i,:)^\ell$ of intermediate matrix $\boldsymbol{S}^\ell$ is given to processor P_m if row $\boldsymbol{A}(i,:)$ and corresponding vertex v_i is assigned to P_m. So, the same row-wise partitioning is induced on matrix $\boldsymbol{S}^\ell$ as with matrices $\boldsymbol{A}$ and $\boldsymbol{G}^\ell$. Matrix $\boldsymbol{S}^\ell$ is computed by following similar steps of computation of $\boldsymbol{Z}^\ell$ in feedforward phase. Then, P_m performs element-wise multiplication, denoted by the $\odot$ operator, to obtain $\boldsymbol{G}_m^{\ell-1} = \boldsymbol{S}_m^\ell \odot \rho'(\boldsymbol{Z}_m^{\ell-1})$.

Algorithm 6.2 gives the proposed parallel backpropagation algorithm. In line 2, each processor P_m computes submatrix $\boldsymbol{G}_m^L$ by using the local vertex-feature matrix $\boldsymbol{H}_m^L$ in the final layer. Here, $\nabla_{\boldsymbol{H}_m^L}\mathbf{J}$ denotes the matrix of partial derivatives of the loss function with respect to $\boldsymbol{H}_m^L$, and its formulation depends on the definition of the loss function. In lines 4–10, matrix $\boldsymbol{S}^\ell$ is computed in a similar way to computation of $\boldsymbol{Z}^\ell$ in Algorithm 6.1. In line 11, gradient matrix $\boldsymbol{G}^{\ell-1}$ for the preceding layer is computed via element-wise multiplication of matrices $\boldsymbol{S}_m^\ell$ and $\rho'(\boldsymbol{Z}_m^{\ell-1})$. In line 12, each processor P_m computes partial results for gradient matrix $\Delta\mathbf{W}^\ell$ of the loss function $\mathbf{J}$ with respect to parameter matrix $\boldsymbol{W}^\ell$.

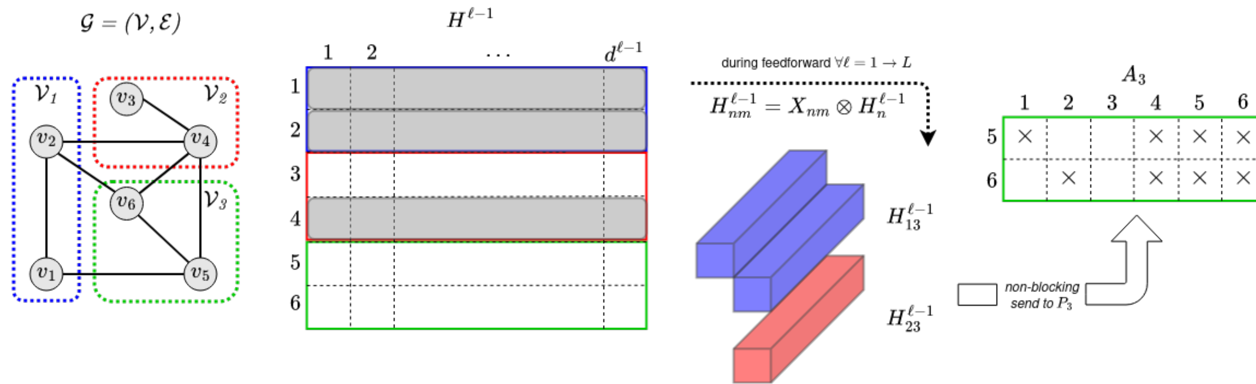
In the computation of $\Delta\mathbf{W}^\ell$, matrix $(\boldsymbol{H}_m^{\ell-1})^T$ is computed in feedforward phase, whereas $(\boldsymbol{A}_m\boldsymbol{G}^\ell)$ part is computed as a by-product in lines 7 and 10. Here, if column $(\boldsymbol{H}^{\ell-1})^T(:,i)$ is stored in $(\boldsymbol{H}_m^{\ell-1})^T$, then the corresponding row $(\boldsymbol{A}\boldsymbol{G}^\ell)(i,:)$ is also stored in $(\boldsymbol{A}_m\boldsymbol{G}^\ell)$. Therefore, multiplication $(\boldsymbol{H}_m^{\ell-1})^T(\boldsymbol{A}_m\boldsymbol{G}^\ell)$ by processor P_m produces matrix $\Delta\mathbf{W}_m^\ell$ of partial products where each nonzero $\Delta\mathbf{W}_m^\ell(i,j)$ contributes to the corresponding nonzero

$$\Delta\mathbf{W}^\ell(i,j) = \sum_m \Delta\mathbf{W}_m^\ell(i,j)$$

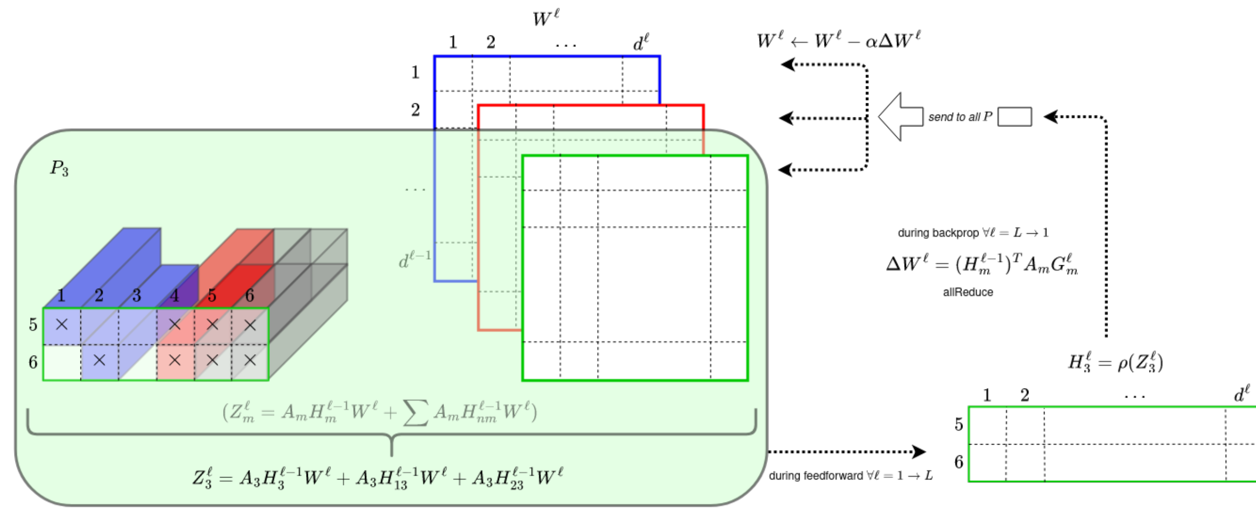
in the final matrix $\Delta\mathbf{W}^\ell$. In line 13, the final gradient matrix $\Delta\mathbf{W}^\ell$ is computed via an allreduce-type communication operation which combines (sums) partial matrices from all processes and distributes the result back to all processes. In line 14, gradient update on $\boldsymbol{W}^\ell$ is performed by all processors on their local copies.

Algorithm 6.2: GCN Parallel Backpropagation

1 **forall** processors P_m **in parallel do**
2 $\quad \boldsymbol{G}_m^L = \nabla_{\boldsymbol{H}_m^L}\mathbf{J} \odot \rho'(\boldsymbol{Z}_m^L)$
3 $\quad$ **for** $\ell = L$ **to** 1 **do**
4 $\quad\quad$ **foreach** $\boldsymbol{X}_{mn} \in \mathcal{S}_m$ **do**
5 $\quad\quad\quad \boldsymbol{G}_{mn}^\ell = \boldsymbol{X}_{mn} \otimes \boldsymbol{G}_m^\ell$
6 $\quad\quad\quad$ Non-blocking send $\boldsymbol{G}_{mn}^\ell$ to processor P_n
7 $\quad\quad \boldsymbol{S}_m^\ell = \boldsymbol{A}_m\boldsymbol{G}_m^\ell(\boldsymbol{W}^\ell)^T$
8 $\quad\quad$ **foreach** $P_n \in \mathcal{R}_m$ **do**
9 $\quad\quad\quad$ Receive $\boldsymbol{G}_{nm}^\ell$ from processor P_n
10 $\quad\quad\quad \boldsymbol{S}_m^\ell = \boldsymbol{S}_m^\ell + \boldsymbol{A}_m\boldsymbol{G}_{nm}^\ell(\boldsymbol{W}^\ell)^T$
11 $\quad\quad \boldsymbol{G}_m^{\ell-1} = \boldsymbol{S}_m^\ell \odot \rho'(\boldsymbol{Z}_m^{\ell-1})$
12 $\quad\quad \Delta\mathbf{W}_m^\ell = (\boldsymbol{H}_m^{\ell-1})^T(\boldsymbol{A}_m\boldsymbol{G}^\ell)$
13 $\quad\quad \Delta\mathbf{W}^\ell =$ Allreduce-sum($\Delta\mathbf{W}_m^\ell$)
14 $\quad\quad \mathbf{W}^\ell \leftarrow \mathbf{W}^\ell - \alpha\Delta\mathbf{W}^\ell$



(a) For graph $\mathcal{G}$, the feature matrix $\boldsymbol{H}^{\ell-1}$ for layer $\ell - 1$ has been conformably partitioned along with $\boldsymbol{A}$ while the weight matrix $\boldsymbol{W}^{\ell}$ has been duplicated across all 3 processors. Processor P_3 (in green) requires $\boldsymbol{H}_{13}^{\ell-1}$ and $\boldsymbol{H}_{23}^{\ell-1}$ from the other two processors (in blue and red respectively).



(b) Computations are demonstrated here on the processor P_3 (in green) which stores $\boldsymbol{A}_3$ and $\boldsymbol{H}_3^{\ell-1}$ locally but must receive $\boldsymbol{H}_{13}^{\ell-1}$ and $\boldsymbol{H}_{23}^{\ell-1}$. The resulting $\boldsymbol{H}_3^{\ell}$ features are similarly computed for all layers ($\ell = 1 \rightarrow L$) during the feedforward phase. Subsequently, during backpropagation ($\ell = L \rightarrow 1$), these are used to generate $\Delta \boldsymbol{W}^{\ell}$ for updation of the weight matrices $\boldsymbol{W}^{\ell}$ on all processors.

Figure 6.1: Communication and computation processes during feedforward and backpropagation phases of the distributed GCN algorithm

Example 6.2.1. A sample execution of the feedforward phase is illustrated in Figure 6.1. The adjacency matrix $\boldsymbol{A}$, and feature matrix $\boldsymbol{H}^{\ell}$ for each layer ℓ are conformably partitioned among the three processors. Thus, each processor P_m only stores submatrices $\boldsymbol{A}_m$ and $\boldsymbol{H}_m^{\ell-1}$. For instance, the computation of matrix $\boldsymbol{Z}_3^{\ell}$ by processor P_3 (in green) requires the other two processors to send $\boldsymbol{H}_{13}^{\ell-1}$ and $\boldsymbol{H}_{23}^{\ell-1}$ corresponding to nonzero indices, and local matrix multiplication is performed without waiting for the completion of these non-blocking communications, to compute $\boldsymbol{H}_3^{\ell}$. Processor P_3 retrieves features of v_1 and v_4 for convolution on vertex v_5, and retrieves features of v_2 and v_4 for vertex v_6. Hence, $\boldsymbol{H}_{13}^{\ell-1}$ contains rows 1 and 2 of $\boldsymbol{H}^{\ell-1}$ while $\boldsymbol{H}_{23}^{\ell-1}$ contains row 4, since these are the nonzero indices of $\boldsymbol{A}_m$ where the feature matrix rows are not locally stored. Note that row 4 is only transferred once to avoid a redundant communication.
Figure 6.1 also displays the computations performed in the backpropagation phase. As seen in the figure, the relatively smaller-sized weight matrices $\boldsymbol{W}^{\ell}$ for each layer ℓ are replicated among all processors. The computation of matrix $\boldsymbol{S}^{\ell}$ is identical to the computation of matrix $\boldsymbol{H}^{\ell}$ and requires the same communication steps which are determined by the partitioning on the adjacency matrix $\boldsymbol{A}$. Matrix $\boldsymbol{S}^{\ell}$ is used together with matrix $\boldsymbol{Z}^{\ell-1}$ to compute gradient matrix $\boldsymbol{G}^{\ell-1}$. The figure also shows the all-reduce operation performed on locally computed matrices $\Delta\boldsymbol{W}_m^{\ell}$ to compute the final matrix $\Delta\boldsymbol{W}^{\ell}$ for gradient update operations.

Extension to GNNs

The main difference between general GNN models [92, 158, 169] and GCNs is in the way messages are created and combined between vertices. In some GNN models, DMM is performed first and messages are created, which is followed by a specialised SpMM for message passing and combining. For example, in a GAT [158], first each vertex feature is transformed with a local parameter matrix (i.e., DMM), and the resulting feature is transmitted to neighbour vertices using the same communication pattern as in SpMM. At the destination vertex, features are concatenated and then multiplied with an attention vector. That is, the order of SpMM and DMM can be changed and additional mathematical operations can be applied to their outputs, without affect the message directions and communication patterns between vertices. Therefore, the proposed partitioning methods can be directly used for other GNN models, and simple modifications to the proposed GCN algorithm can support the additional computations necessary alongside this communication scheme.

6.3 Partitioning models for full-batch training

The use of point-to-point communication operations in the proposed solution enables communication to be reduced further via sparse matrix partitioning strategies [29]. A sparse matrix partitioning scheme is developed to distribute the adjacency, vertex-feature, and gradient matrices used in computations among processors, based on a hypergraph partitioning model for the original graph.

Different partitioning models may be used for splitting the adjacency matrix among processors. In this chapter, it is shown that, during the message-passing operations, the hypergraph partitioning model encodes SpMM communication costs more accurately than the graph partitioning model which is in popular use (e.g., in DistDGL [194]).

Graph model

In a graph model, a p-way partitioning $\Pi_p = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_p\}$ over vertex set $\mathcal{V}$ induces a row-wise partitioning on matrix $\boldsymbol{A}$ among p processors. If a vertex v_i is assigned to part $\mathcal{V}_m \in \Pi_p$, then row $\boldsymbol{A}(i,:)$ is assigned to processor P_m. Note that the input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ in GCN training can be directed or undirected, but graph partitioning tools (e.g., METIS) assume that the graph to be partitioned is undirected, edges have integer costs, and vertices have integer weights.

Under a partition Π_p, each undirected cut edge (v_i, v_j) represents the communication of $\boldsymbol{H}^{\ell-1}(i,:)$- and $\boldsymbol{H}^{\ell-1}(j,:)$-matrix rows between respective processors during feedforward phase, and communication of $\boldsymbol{G}^{\ell}(i,:)$- and $\boldsymbol{G}^{\ell}(j,:)$-matrix rows during backpropagation phase. Since there is a d-dimensional vertex feature matrix $\boldsymbol{H}^{\ell} \in \mathbb{R}^{n \times d_\ell}$, each undirected edge encodes a total communication volume of $\sum_\ell 2(d_{\ell-1} + d_\ell)$ nonzero entries over all layers ℓ. Because the communication volume encoded by each edge is the same constant value, each undirected edge (v_i, v_j) can be associated with a unit cost$(v_i, v_j) = 1$. Each vertex v_i is associated with a computational weight $w(v_i) = |\text{cols}(\boldsymbol{A}(i,:))|$. DistDGL [194] also utilises this partitioning scheme and partitions the input graph via METIS, only considering undirected graphs.

The graph model is less accurate compared to the hypergraph model because the former overestimates the total communication volume between processors. This deficiency of the graph model can be seen in two ways: i) When both of the directed edges (v_i, v_j) and (v_j, v_i) are not simultaneously present in the input graph $\mathcal{G}$, the graph model still considers an undirected edge (v_i, v_j) that sees communication in both ways although the communication is actually one-way. ii) If a vertex v_i is connected to vertices v_j and v_ℓ that are stored together but on a different processor from v_i, the graph model assumes that the features of v_i are sent twice. However, these features are sent to that processor once in a single message. These two cases cause the partitioning cut size to be higher than the actual communication volume when the graph model is employed.

Hypergraph model

1D row-wise partitioning of adjacency matrix can be modelled as a hypergraph partitioning problem [26] since the hypergraph model can encode the exact communication volume of parallel GCN. The connectivity cut size of the hypergraph model encodes the total communication volume among processors, while weights of partitions encode the associated computational load for processors. Hence, minimisation of the connectivity cut size under weight balancing constraints achieves minimisation of the total communication volume while achieving computational load balance. During the feedforward phase, the hypergraph model encodes the total communication volume on $\boldsymbol{H}^{\ell-1}$-matrix rows for parallel SpMMs $\boldsymbol{A}\boldsymbol{H}^{\ell-1}$ among processors in each layer ℓ. The model also encodes the total communication volume on $\boldsymbol{G}^{\ell}$-matrix rows for parallel SpMMs $\boldsymbol{A}\boldsymbol{G}^{\ell}$ during backpropagation phase.

To partition adjacency matrix $\boldsymbol{A}$, first a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is built where for each matrix row $\boldsymbol{A}(i,:)$ there exists one vertex $v_i \in \mathcal{V}$ and for each column $\boldsymbol{A}(:,j)$, there exists one net $n_j \in \mathcal{N}$. Similar to the graph model, a partitioning obtained on the vertex set of

the input graph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ also induces a 1D row-wise partitioning on the adjacency matrix. That is, a p-way partitioning $\Pi_p = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_p\}$ over vertex set $\mathcal{V}$ induces a row-wise partitioning on matrix $\boldsymbol{A}$ among p processors, since each vertex v_i corresponds to row $\boldsymbol{A}(i,:)$. Additionally, each vertex $v_i \in \mathcal{V}$ also represents the task of computing rows $\boldsymbol{Z}^\ell(i,:)$ and $\boldsymbol{S}^\ell(i,:)$ in each layer ℓ. Therefore, each vertex v_i is associated with weight $w(v_i) = |\text{cols}(\boldsymbol{A}(i,:))|$, i.e., the number of nonzero column indices in the ith row of matrix $\boldsymbol{A}$, to encode the computational load of the task represented by vertex v_i. Note that the number of nonzero arithmetic operations required to compute rows $\boldsymbol{Z}^\ell(i,:)$ and $\boldsymbol{S}^\ell(i,:)$ is proportional to the number of nonzero column indices in row $\boldsymbol{A}(i,:)$. So, satisfying the balancing constraints in hypergraph partitioning achieves computational load balance.

Net set $\mathcal{N}$ encodes task dependencies on rows of matrices $\boldsymbol{H}^{\ell-1}$ and $\boldsymbol{G}^\ell$ during feedforward and backpropagation phases for each layer ℓ. Each net $n_j \in \mathcal{N}$ connects all vertices $v_i \in \mathcal{V}$ for which the corresponding row $\boldsymbol{A}(i,:)$ has a nonzero entry in the jth column. For computing rows $\boldsymbol{Z}(i,:)^\ell$ and $\boldsymbol{S}(i,:)^\ell$, the processor that owns row $\boldsymbol{A}(i,:)$ needs all $\boldsymbol{H}^{\ell-1}$- and $\boldsymbol{G}^\ell$-matrix rows, corresponding to nonzero column indices $\text{cols}(\boldsymbol{A}(i,:))$, respectively. Therefore, pins of a net n_j denotes the tasks that require row $\boldsymbol{H}^{\ell-1}(j,:)$ and $\boldsymbol{G}^\ell(j,:)$. Formally, pins of a net n_j can be written as

$$\text{pins}(n_j) = \{v_i \in V \mid \exists j \in \text{cols}(\boldsymbol{A}(i,:))\}.$$

Under a partitioning Π_p, a net $n_j \in \mathcal{N}$ with connectivity set $\Lambda(n_j)$ encodes the total communication volume on rows $\boldsymbol{H}^{\ell-1}(j,:)$ and $\boldsymbol{G}^\ell(j,:)$ in each layer ℓ. Here, at least one part in $\Lambda(n_j)$ stores vertex v_j since each diagonal entry contains a nonzero entry in adjacency matrix $\boldsymbol{A}$. That is, for all net $n_j \in \mathcal{N}$, vertex $v_j \in \text{pins}(n_j)$. Therefore, a part $\mathcal{V}_m \in \Lambda(n_j)$ stores vertex v_j and hence, processor P_m stores rows $\boldsymbol{H}^{\ell-1}(j,:)$ and $\boldsymbol{G}^\ell(j,:)$ in its local submatrices $\boldsymbol{H}_m^{\ell-1}$ and $\boldsymbol{G}_m^\ell$, respectively. Due to the task dependencies encoded by net n_j, processor P_m sends row $\boldsymbol{H}^{\ell-1}(j,:)$ to all processors corresponding to parts in $\Lambda(n_j) \backslash \mathcal{V}_m$ during feedforward phase, i.e., $\lambda(n_j) - 1$ communications. Similarly, processor P_m sends row $\boldsymbol{G}^\ell(j,:)$ to all processors in $\Lambda(n_j) \backslash \mathcal{V}_m$ during backpropagation phase. If a processor $P_n \in \Lambda(n_j) \backslash \mathcal{V}_m$ has multiple vertices connecting to net n_j, processor P_n receives row $\boldsymbol{H}^{\ell-1}(j,:)$ and row $\boldsymbol{G}^\ell(j,:)$ only once. So, net n_j incurs a communication volume of $\text{cost}(n_j) \times (\lambda(n_j) - 1)$ where the cost of net n_j is denoted as $\text{cost}(n_j) = \sum_\ell d_{\ell-1} + d_\ell$ since all nonzero entries in rows $\boldsymbol{H}^{\ell-1}(j,:)$ and $\boldsymbol{G}^\ell(j,:)$ are communicated in each layer ℓ. Since the cost of each net is the same constant value, each net can also be associated with a unit cost $cost(n_j) = 1$. Therefore, the total communication volume $\sum_{n_j \in \mathcal{N}} 2 \times \text{cost}(n_j) \times (\lambda(n_j) - 1)$ can be written as

$$\sum_{n_j \in \mathcal{N}} 2 \times (\lambda(n_j) - 1)$$

which indicates that minimising the connectivity cut size corresponds to minimising the total communication volume.

Example 6.3.1. Figure 6.2 displays an illustrative example of the proposed hypergraph partitioning model on a sample graph $\mathcal{G}$ having adjacency matrix $\boldsymbol{A}$. The hypergraph $\mathcal{H}$ is constructed having parts $\mathcal{V}_1$ (blue), $\mathcal{V}_2$ (red), and $\mathcal{V}_3$ (green) each containing two vertices,

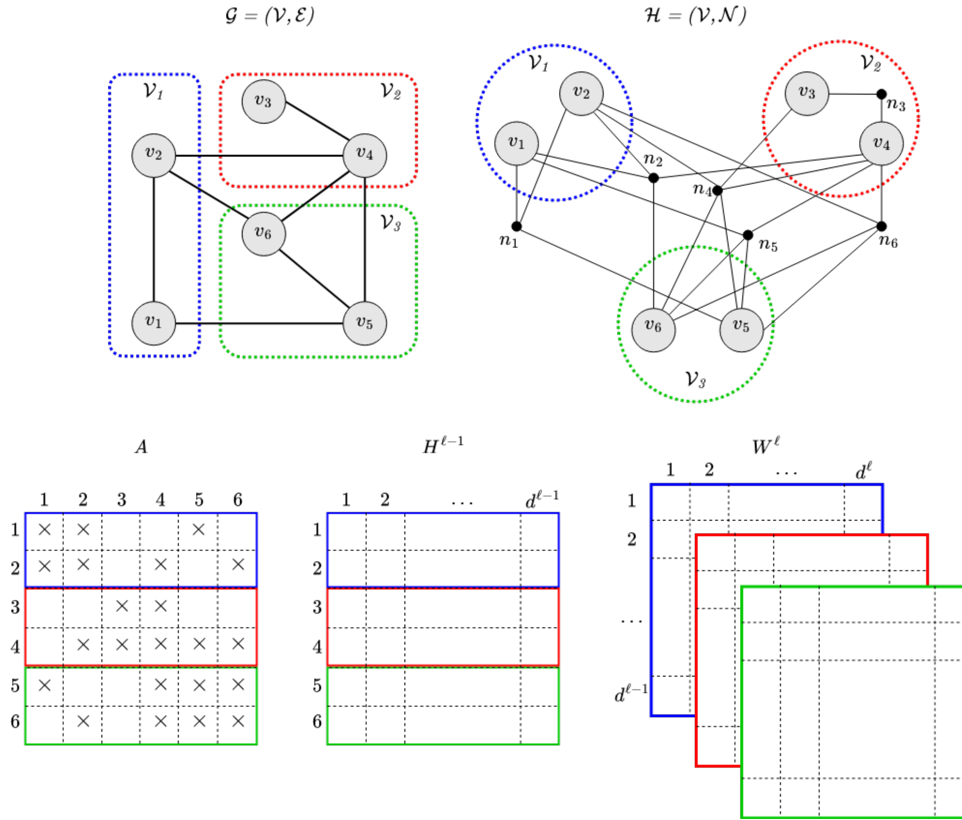


Figure 6.2: Hypergraph partitioning of graph $\mathcal{G}$ having adjacency matrix $\boldsymbol{A}$ (including self loops), by constructing the corresponding hypergraph $\mathcal{H}$ where every net n_j connects nonzero entries of the column i in $\boldsymbol{A}$. The feature matrix $\boldsymbol{H}^{\ell}$ for layer ℓ is conformably partitioned along with $\boldsymbol{A}$, while the weight matrix $\boldsymbol{W}^{\ell}$ is duplicated across all processors.

with a net n_j for every column $\boldsymbol{A}(:, j)$. According to the hypergraph partitioning model, rows of $\boldsymbol{A}$ are assigned to processors based on the hypergraph vertex partitioning. For example, row $\boldsymbol{A}(i, :)$ will be stored on processor P_1 as vertex (v_1), and is assigned to $\mathcal{V}_1$. Since (v_1) represents a task, the computational load is proportional to the number of non-zero columns in row 1 and is encoded by its weight $w(v_1) = 3$. Each net connects non-zero entries in a row. For example net n_2 connects $pins(n_2) = \{(v_1), (v_2), (v_4), (v_6)\}$ with connectivity set $\Lambda(n_2) = \{\mathcal{V}_1, \mathcal{V}_2, \mathcal{V}_3\}$. Its connectivity is therefore $\lambda(n_2) = 3$. The net set $\mathcal{N}$ thus encodes task dependencies during the feedforward and backpropagation phases since communication operations on matrices $\boldsymbol{H}^{\ell}$ and $\boldsymbol{G}^{\ell-1}$ are identical and determined by the partitioning on matrix $\boldsymbol{A}$. The feature matrix $\boldsymbol{H}^{\ell-1}$ is conformably partitioned with $\boldsymbol{A}$ for each layer of the GCN, while the weight matrix $\boldsymbol{W}^{\ell}$ is replicated across the three processors.

Figure 6.2 also depicts how the graph model overestimates the communication volume. Features of vertex (v_4) must be fetched by vertices (v_2), (v_3), (v_5), and (v_6). According to the graph model, the feature vector of (v_4) is encoded as if it were sent from processor P_2 to processor P_3 twice, but should only be sent once. Therefore, cut edges connecting to vertex (v_4) in the graph encode a communication volume of 3 instead of the true value of 2. On the other hand, the hypergraph model shown on $\mathcal{H}$ uses net n_4 to encode communications from vertex (v_4). Since the connectivity of n_4 is $\lambda(n_4) = 3$ and hypergraph partitioning minimises connectivity-1 metric, net n_4 encodes the true communication volume as $\lambda(n_4) - 1 = 2$.

6.4 Partitioning model using hypergraph sampling for mini-batch training

The hypergraph/graph models described in the previous section encode the communication volume in full-batch training. In mini-batch training, some chosen sampling technique is applied to the input graph to produce subgraphs on which convolutions are performed.

Stochastic hypergraph model

A stochastic hypergraph model is proposed which encodes and minimises the *expected* communication volume in mini-batch training. The partitioning model is applicable alongside any desired sampling technique being used for the generation of mini-batches, e.g., GraphSAGE neighbourhood sampling.

First, mini-batches (i.e., subgraphs) are randomly generated using any sampling technique. Next, for each subgraph, a hypergraph is constructed that encodes the total communication volume for the mini-batch. By merging all hypergraphs generated (one per mini-batch), a larger hypergraph is built that can encode the *expected* connectivity of any randomly generated net. Partitioning the resulting merged stochastic hypergraph minimises the expected connectivity of a random net, and thus minimises the expected total communication volume for any randomly generated mini-batch. Additionally, if each vertex is equally likely to be selected in a mini-batch, then the same vertex weighting and balancing constraint in the hypergraph model for full-batch training can be applied here to achieve computational load balance for mini-batch training.

More formally, given an input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, each mini-batch corresponds to a subgraph $\mathcal{G}' = (\mathcal{V}' \subset \mathcal{V}, \mathcal{E}' \subset \mathcal{E})$. The proposed stochastic hypergraph partitioning process is described in Algorithm 6.3. First, b mini-batches are generated (line 1, each corresponding to a subgraph $\mathcal{G}'_i = (\mathcal{V}'_i, \mathcal{E}'_i)$ for $i = 1, 2, \ldots, b$. For each such subgraph $\mathcal{G}'$ a hypergraph $\mathcal{H}' = (\mathcal{V}', \mathcal{N}')$ is built in the same way as in full-batch training (line 2). In line 3, the stochastic hypergraph $\mathcal{H} = (\mathcal{V} = \bigcup \mathcal{V}'_i, \mathcal{N} = \bigcup \mathcal{N}'_i)$ is formed by merging all vertex and net sets into the corresponding sets for the merged hypergraph. Finally, a partitioning Π of $\mathcal{H}$ is computed which can be used to determine the row-wise partitioning of the adjacency matrix (line 4).

Algorithm 6.3: Stochastic Hypergraph Partitioning

1 Generate b subgraphs $\mathcal{G}_i = (\mathcal{V}'_i, \mathcal{E}'_i)$ of $\mathcal{G}$ for $i = 1, 2, \ldots, b$
2 Build hypergraph $\mathcal{H}_i = (\mathcal{V}'_i, \mathcal{N}'_i)$ for each $\mathcal{G}_i = (\mathcal{V}'_i, \mathcal{E}'_i)$
3 Build stochastic hypergraph $\mathcal{H} = (\mathcal{V} = \bigcup_{i=1}^{b} \mathcal{V}'_i, \mathcal{N} = \bigcup_{i=1}^{b} \mathcal{N}'_i)$
4 Partition p-way hypergraph $\mathcal{H}$ to obtain partitioning $\Pi = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_p\}$
5 **Return** Π

Under a p-way vertex partition Π of the stochastic hypergraph $\mathcal{H}$, let λ denote the expected connectivity of a randomly generated net. By using Hoeffding's inequality, the value of λ can be estimated within its ϵ error with a probability of at least $1-\delta$, and is dependent on $|\mathcal{N}|$. That is, if an adequate number of nets are generated, the stochastic hypergraph model encodes the expected communication volume with low error with high probability. Since

the expected connectivity λ is determined by the partitioning Π over the hypergraph, the stochastic hypergraph partitioning can minimise this objective.

6.5 Experimental evaluation

The parallel training algorithm (with different choices of partitioning strategy) is evaluated on numerous datasets to test its performance in terms of scalability, communication costs, running time, and predictive accuracy.

Datasets

The performance of the proposed parallel GCN training algorithm is evaluated on a diverse set of real-world graphs from popular applications that use GCN models such as citation networks, social networks, road networks, and product co-purchasing networks. The first eight datasets (**amazon0601**, **cit-Patents**, **coPapersDBLP**, **com-Amazon**, **com-Youtube**, **flickr**, **roadNet-CA**, and **soc-Slashdot0902**) are from the Stanford Large Network Dataset Collection.[1] **ogbn-Papers100M** is available from the Open Graph Benchmark collection.[2] **reddit** is a dataset hosted by Deep Graph Library (DGL).[3] Properties of these graphs are displayed in Table 6.1.

Table 6.1: Dataset properties

Dataset	#Nodes	#Edges	Type
amazon0601	403,394	3,387,388	Directed
cit-Patents	3,774,768	16,518,948	Directed
coPapersDBLP	540,486	30,491,458	Undirected
com-Amazon	334,863	1,851,744	Undirected
com-Youtube	1,134,890	5,975,248	Undirected
flickr	820,878	9,837,214	Directed
roadNet-CA	1,971,281	5,533,214	Undirected
soc-Slashdot0902	82,168	948,464	Directed
ogbn-Papers100M	111,059,956	1,615,685,872	Directed
Cora	2708	10556	Undirected
reddit	232,965	114,615,892	Undirected

Baselines

DGL (with PyTorch [4] v1.6 backend) implementation of GCN is used as the baseline, and speedup values are computed according to its single-node CPU performance. The proposed solution performance is also compared against **CAGNET** [154] which is the most closely related algorithm. The comparison is performed by using both the original GPU implementation and a custom CPU implementation of CAGNET. Comparisons against Neugraph [112] and Roc [81] are omitted as they are not compatible with CPU clusters, and CAGNET already provides much more scalability.

[1] https://snap.stanford.edu/data/
[2] https://ogb.stanford.edu/docs/nodeprop/
[3] https://docs.dgl.ai/en/0.8.x/api/python/dgl.data.html
[4] https://pytorch.org/

The improvements in performance are evaluated of the proposed parallel GCN training algorithm with both hypergraph partitioning (**HP**) and graph partitioning (**GP**) models used to partition the input matrices. For mini-batch training, the stochastic hypergraph (**SHP**) model is tested. Additionally, results are reported for random partitioning (**RP**) as a baseline, which evenly splits the adjacency matrix by assigning rows to processors uniformly at random, and is a competitive method for balancing computational load and communications.

Setup

CPU experiments are run on a cluster of 180 compute nodes with 2x Intel Xeon Platinum 8268 2.9 GHz 24-core processors (48 cores per node) and 4GB RAM per core. GPU experiments use the Sulis[5] cluster of 30 nodes each with 3x NVIDIA A100 GPUs and 4GB RAM per core. Both use InfiniBand interconnect (100 Gbit/s) and Slurm Workload Manager. The single-node DGL implementation requires a server with a better hardware (memory) configuration, thus a 16-core Intel Xeon 3.90GHz processor is used having 500 GB memory.

The CPU code is in C++, using the GraphBLAS library for local sparse matrix operations and MPI for point-to-point communication operations. The GPU version uses PyTorch with NCCL backend to perform communication operations [9].

PaToH [27] hypergraph partitioning tool and METIS [84] graph partitioning tool are employed. For `ogbn-Papers100M`, KaHyPar [139] is used which can handle massive-scale graphs. The partitioning tools are used with their default parameters and the maximum imbalance ratio is set as $\vartheta = 0.01$.

Communication costs

Table 6.2 compares HP, GP, and RP in terms of the communication volume and message counts metrics they incur on 512 processors (i.e., MPI processes). For each partitioning method, the parallel GCN algorithm is run with random vertex features and label data for five epochs, to measure the communication cost and running time metrics. These metrics respectively relate to bandwidth and latency costs induced by different partitioning strategies on parallelisation costs. In the table, both the average and maximum volume/number of messages sent by a processor are displayed. Average values are proportional to the total message volume/count values, and are used to show how much the maximum values deviate from the mean.

For each input graph, the first and second rows denote the respective values attained by HP and GP, where these values are normalised with respect to values attained by RP. The third row denotes the ratios of values attained by HP and GP (i.e., HP/GP). The first column (i.e., 'R') in the table indicates the ratio of the parallel running time of HP and GP to that of RP. The last column (i.e., 'S') denotes the speedup values attained by HP and GP with respect to single-node running time performance of DGL. At the end of the table, the geometric means of the normalised values for HP and GP are given along with the ratios of these values in the last row. For instance, the 'R' and 'S' columns for `amazon0601` are interpreted as follows: The parallel running times of HP and GP divided by that of RP is 0.63 and 0.65, respectively.

[5] `https://sulis-hpc.github.io/`

Table 6.2: Performance comparison with HP, GP, and RP on $p = 512$ CPUs

			Volume		Messages		
		R	Avg	Max	Avg	Max	S
amazon0601	HP	0.63	0.12	0.29	0.22	0.51	10.88
	GP	0.65	0.18	0.31	0.30	0.62	10.55
	HP/GP	0.97	0.67	0.92	0.74	0.82	
cit-Patents	HP	0.77	0.17	0.29	0.70	0.89	8.48
	GP	0.80	0.19	0.50	0.77	0.94	8.10
	HP/GP	0.95	0.88	0.57	0.91	0.95	
coPapersDBLP	HP	0.32	0.07	0.08	0.42	0.71	10.93
	GP	0.69	0.07	0.16	0.57	0.77	5.04
	HP/GP	0.46	0.97	0.49	0.74	0.92	
com-Amazon	HP	0.32	0.09	0.20	0.14	0.32	14.31
	GP	0.37	0.14	0.27	0.19	0.42	12.37
	HP/GP	0.86	0.60	0.73	0.72	0.75	
com-Youtube	HP	0.40	0.36	0.52	0.72	0.97	10.85
	GP	1.45	0.37	2.60	0.90	0.99	3.01
	HP/GP	0.28	0.98	0.20	0.81	0.98	
flickr	HP	0.81	0.45	0.60	0.79	1.00	9.59
	GP	11.13	0.38	6.89	0.96	1.00	0.70
	HP/GP	0.07	1.19	0.09	0.82	1.00	
roadNet-CA	HP	0.19	0.01	0.01	0.01	0.03	30.32
	GP	0.20	0.01	0.02	0.01	0.03	29.08
	HP/GP	0.96	0.78	0.67	1.03	1.00	
soc-Slashdot0902	HP	0.75	0.74	0.69	0.86	0.92	3.50
	GP	2.02	0.85	4.38	0.93	1.00	1.30
	HP/GP	0.37	0.86	0.16	0.92	0.92	
	mean HP	0.47	0.13	0.21	0.29	0.48	10.60
	mean GP	0.98	0.15	0.56	0.35	0.52	5.04
	mean HP/mean GP	0.48	0.87	0.37	0.83	0.92	

The parallel running time of HP divided by that of GP is 0.97. Speedups achieved by HP and GP with respect to DGL, displayed under column 'S', are 10.88 and 10.55, respectively.

As seen in Table 6.2, both HP and GP provide significant improvements over communication volume and message count metrics. On average, HP and GP incur 87% and 85% less average communication volume than RP respectively, with HP performing 15% better than GP. In terms of maximum communication volume, HP consistently outperforms RP, providing 79% improvement on average over RP. HP performs 63% better than GP and provides better communication balance. Even though GP provides 44% improvement on average over RP, its performance significantly degrades for graphs `com-Youtube`, `flickr`, and `soc-Slashdot0902` where for instance, GP performs 6.89x worse than RP for `flickr`. Although both partitioning methods provide significant improvement in average communication volume, graph partitioning can disrupt the communication balance between processors. For the message count metrics, on average, HP and GP reduce the total number of messages by 71% and 65% as compared to RP while HP performs 17% better than GP. Similarly, maximum message count is respectively reduced by 52% and 48% by HP and GP while HP performs 9% better than GP.

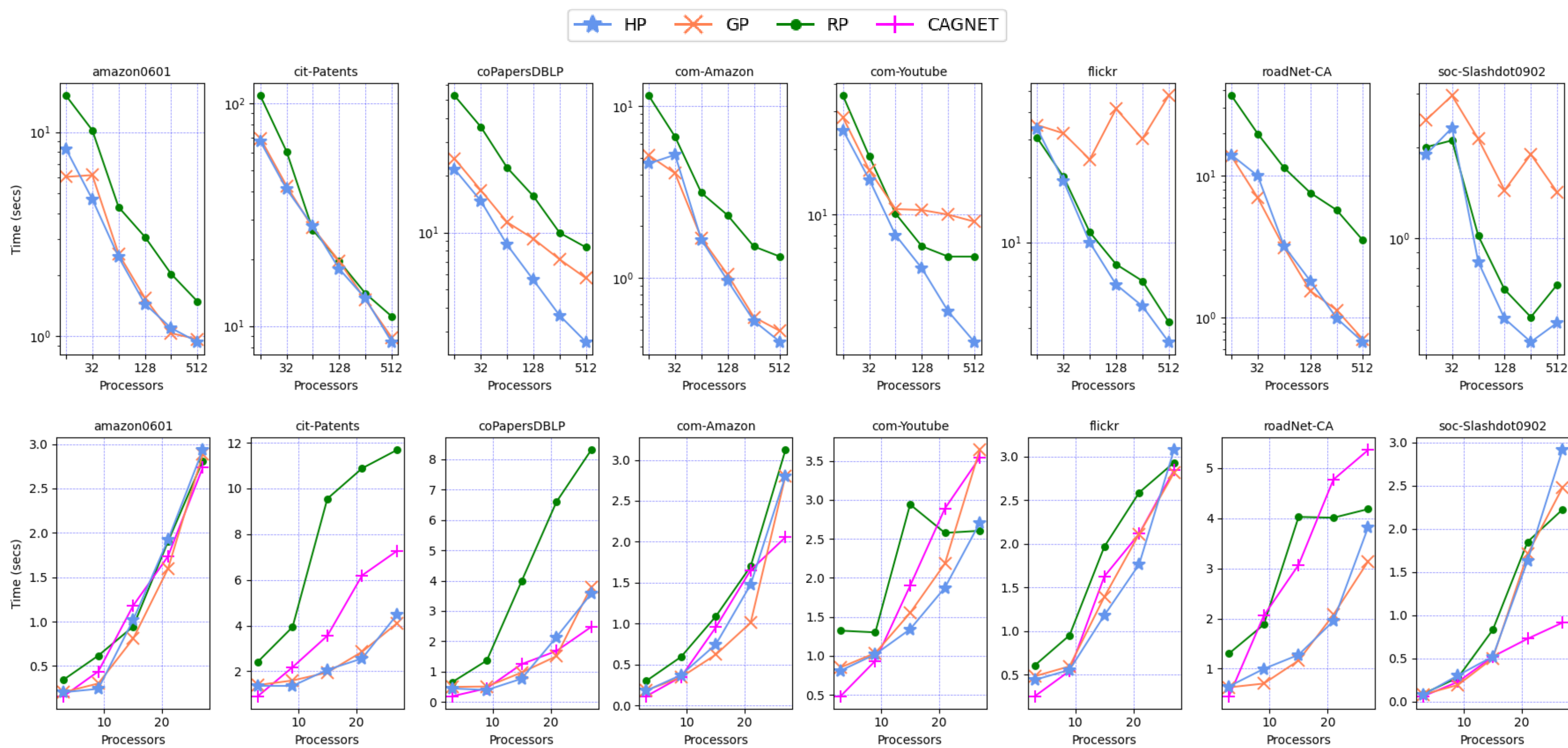


Figure 6.3: Strong scaling for full-batch training with HP, GP, and RP for $p = 16$ to $p = 512$ CPUs (top row) and with HP, GP, RP, and CAGNET (CN) for $p = 3$ to $p = 27$ GPUs (bottom row)

Parallel running time efficiency

Improvements in communication costs by HP and GP considerably reduce parallel running of RP. On all graphs, HP provides an average of 2.12x speedup over RP, despite the latter achieving good communication and computation balance. GP runs 20%–80% faster than RP for most graphs. However, it is slower than RP on `flickr`, `com-Youtube` and `soc-Slashdot0902`. The reason for this is the communication volume imbalance, as can be seen from the maximum and average communication volumes achieved on these graphs. The best performance is achieved by HP and GP for `roadNet-CA` where both partitioning methods provide approximately 99% improvement in communication volume and message count metrics compared to RP and run 5x faster.

As seen in the speedup column, on average, HP and GP provide 10.60x and 5.04x speedup respectively over the DGL implementation. The best speedup is achieved for `roadNet-CA` where HP and GP provide 30.32x and 29.08x speedup. This is because road networks are relatively more sparse as compared to the other social networks and hence the amount of data transferred between processors reduces in such cases. As the graph sizes increase and graphs become more sparse, partitioning tools usually perform better optimisations.

Figure 6.3 displays strong scaling of HP, GP, and RP on CPU (first row) and GPU (second row) clusters. As seen here, on the CPU cluster, HP achieves almost linear speedup up to 512 cores on all input graphs. Additionally, HP either matches or outperforms GP, and always outperforms RP. The reason for the speedup loss of HP on 512 processors for `soc-Slashdot0902` is the relatively smaller size of the graph as compared to the others. In general, GP performs better than RP except for `flickr`, `com-Youtube`, and `soc-Slashdot0902` graphs due to degradation of communication balance as also shown in Table 6.2. Here, line plots for the CPU implementation of CAGNET are omitted since HP and GP significantly outperform it.

Algorithms that are focused on optimising communication operations are demonstrated to be better suited to CPU clusters over GPUs, and where the sparsity of the problem is important to exploit. As seen in Figure 6.3, for GPU cluster experiments, the PyTorch implementations (with NCCL backend) of both CAGNET and the proposed parallel GCN (i.e. HP, GP and RP) do not scale well (up to 27 GPUs). The reason for this is the high communication efficiency needed to attain speedup on GPUs. In comparison to MPI, the NCCL backend cannot provide the necessary efficiency. On GPUs, the proportion of total running time that is spent on local computation is small, therefore the gains obtained via parallelisation do not amortise the time spent for communication on larger GPU counts. In addition, despite the optimisations obtained in the communication volume from the proposed algorithm, with the NCCL backend these are not as effective as with MPI. This results in limited performance improvement on the overall parallel run time due to the higher latency costs. HP and GP continue to be faster than CAGNET for most datasets and settings. In addition, the performance improvement is expected to be more stark at higher GPU counts, as can be seen from the CPU cluster results, but no suitable larger GPU clusters were available for experiments. Moreover, the parallel CPU implementation for HP is found to outperform the GPU version in many cases. For example, on `amazon0601`, the running time is 0.94 seconds on 512 CPUs, while

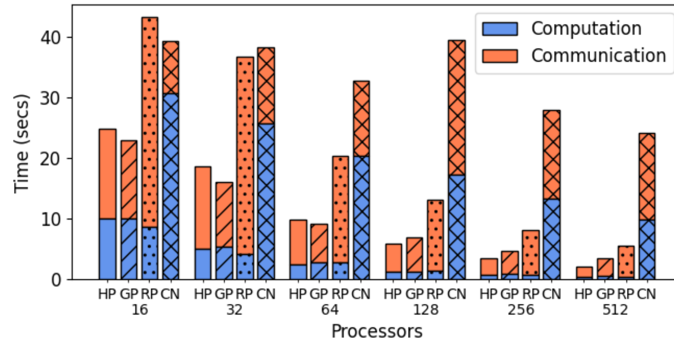


Figure 6.4: Performance comparisons for full-batch training. Communication time and computation time split with HP, GP, RP, and CAGNET (CN) on coPapersDBLP for $p = 16$ to $p = 512$ CPUs

on 15 GPUs it takes as long as 1.02 seconds. On the larger roadNet-CA dataset, the same setting takes only 0.67 seconds on CPU and twice as long (1.28 seconds) on GPU.

Communication and computation time comparison

Figure 6.4 analyses the breakdown of communication and local computation times in the total parallel CPU running time of HP, GP, RP, and CAGNET (CN) for coPapersDBLP. On all processor counts, HP and GP consistently perform better than CAGNET, with HP being the best method at high processor counts. On the largest processor count, HP runs nearly 12x faster than CAGNET. Even though RP performs worse than CAGNET on $p = 16$ processors, its performance becomes better as the number of processors increases. As seen in the figure, the total communication time decreases with the total computation time for HP, GP and RP as the number of processors increases, whereas the communication time of CAGNET increases. This is because point-to-point communication necessitates each processor to communicate only with a small subset of processors and thus incurs lower communication volume and latency costs, whereas broadcast communication involves all processors and incurs higher communication overheads due to the unnecessary data and message transfer. The redundant computations in CAGNET are also visible in its higher local computation times. Moreover, better optimisations are achieved by HP than GP, which is evident from the communication time of GP being 1.7x higher and CAGNET being 8.3x higher than that of HP on 512 processors. HP shows between 2.4x to 3x better communication efficiency over RP from low processor counts to high, while the communication benefit of GP over RP drops slightly from 2.7x to 1.8x.

Scalability for deeper networks

Figure 6.5 shows speedup performance of HP, GP, and RP when varying the number of layers and the dimensionality of features, on roadNet-CA for 512 CPUs. The number of dimensions is chosen as $d = 50$ and $d = 100$, and the number of layers is increased from 2 to 8. The speedup is computed by dividing the running time of DGL by the running time of HP, GP, or RP respectively under the same GCN configuration. When the number of layers increases and d is kept constant, there is no loss of speedup in any algorithm, and speedup in fact increases for HP. The speedups decrease as d increases because of the rise

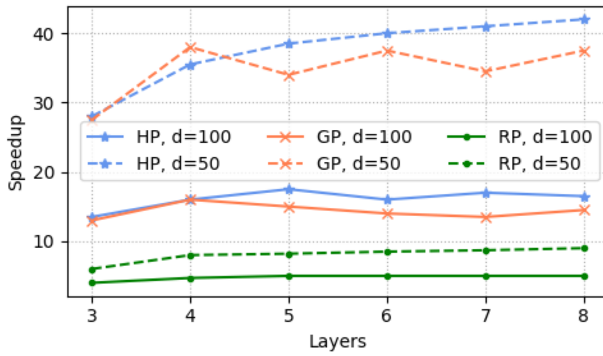


Figure 6.5: Performance comparisons for full-batch training. Speedup with increasing layers ($L = 3, 4, \ldots, 8$) and dimensions ($d = 50, 100$) on `roadNet-CA` for $p = 512$ CPUs

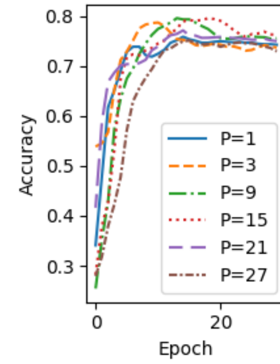


Figure 6.6: GNN model accuracy with HP on `Cora` for $p = 1$ to $p = 27$ GPUs

in total communication volume, which reduces the parallelisation efficiency. For example, the speedup of HP decreases approximately from 40x to 17x when the number of features is increased from $d = 50$ to $d = 100$, for an 8-layer GCN. On the other hand, the performance of HP increases from approximately 28x to 40x when the number of layers is increased from 3 to 8, for $d = 100$.

Predictive performance

The effect on predictive performance of the GCN model is also examined when parallelised using the proposed training algorithm. The downstream task studied here is node classification. The `Cora` dataset is used since the large-scale networks do not have training labels. The parallel training algorithm is run for 30 epochs on up to 27 GPUs, and its accuracy performance is compared with the serial training algorithm. Figure 6.6 shows that the parallel training algorithm does not have any negative impact on the accuracy performance, with approximately 75% accuracy achieved in all settings using hypergraph partitioning. The GCN model performs exact full-batch computations across all the parallelised settings, and the predictive performance remains unchanged within some tolerance (due to the stochastic nature of the training steps).

Stochastic hypergraph model

Figure 6.7 shows the relative performance improvement of the stochastic hypergraph model (SHP) over HP in mini-batch training. 10K random mini-batches of size 20K vertices are generated. The total communication volume they induce under partitionings obtained by HP and SHP on `com-Amazon` graph using GPU are measured. $\epsilon = 0.1$ and $\delta = 0.5$ are used to run SHP and the same maximum imbalance ratio ($\vartheta = 0.01$) is set for both SHP and HP. As seen in the figure (inset), where the relative improvement (in red) of SHP over HP is reported along the secondary y-axis, HP induces 10% more communication volume than SHP on average. The performance difference in favour of SHP is even more pronounced at higher processor counts. Furthermore, SHP is seen to provide a greater benefit of shorter running time with more processors.

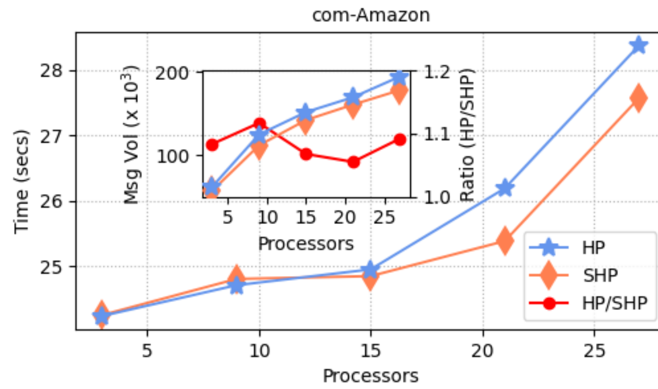


Figure 6.7: Performance comparisons for mini-batch training. Running time and communication volume ('Msg Vol') with HP and SHP on `com-Amazon` for $p = 3$ to $p = 27$ GPUs

Scalability to billion-scale datasets

The algorithm is also tested on a billion-scale `ogbn-Papers100M` dataset, which is only feasible when partitioned onto 27 GPUs due to memory limitations. Table 6.3 shows that the method is scalable not only to high processor counts but also to very large graphs. RP slows down significantly with increasing dimensionality of features. On the other hand, the communication benefit of HP, reducing communication volume approximately by a factor of 10x, allows it to scale better.

Table 6.3: Performance comparison with HP and RP ($d = 1, 2, 5$) on `ogbn-Papers100M` for $p = 27$ GPUs

Partitioning model	Running time (secs) $d = 1$	$d = 2$	$d = 5$	Communication volume
HP	24.46	25.00	29.73	1.2 billion
RP	34.70	42.88	65.14	13 billion

Comparison against SOTA

The optimisations are proposed for large-scale CPU clusters, since the improvement of point-to-point communication overheads over broadcast is more pronounced in such cases. Nonetheless the running time of the GPU implementation (HP) is compared against state-of-the-art (SOTA) distributed GPU systems. All systems use the same GCN architecture and report results on the `reddit` dataset that is common among them. The reported algorithms (except CAGNET) use methods that affect training and predictive performance, such as caching, vertex replication, and asynchronous parameter updates, whereas HP performs full-batch training with negligible impact on accuracy. As seen from the results, HP achieves considerable relative performance even on small GPU counts.

6.6 Discussion

In this chapter, a highly parallel algorithm is developed for GCN training on large-scale distributed-memory systems. For scalability, all matrices except parameter matrices are

Table 6.4: Running time (in seconds per epoch) compared to SOTA on `reddit`

Method	Running time (per epoch)	Setup	Reference
HP	0.67	A100*3	-
CAGNET	0.11	V100*4	Fig 1 (c=1) [154]
ROC	1/5 = 0.20	P100*4	Fig 5 [81]
Sancus	97.4/1000 = 0.09	V100*4	Table 4 (SCS-A) [127]
PaGraph	≈ 1.00	1080Ti*1	Fig 9 [106]
Dorylus	162.9/120 = 1.36	V100*2	Fig 5, Table 4 [152]
DGCL	0.15	V100*4	Fig 8(a) [24]

row-wise partitioned between processors. The algorithm achieves further communication cost reduction by capturing the sparsity pattern of the adjacency matrix to perform point-to-point communications, via the use of a sparse matrix partitioning scheme based on an intelligent hypergraph model. This is especially useful when the size of input graphs make using shared-memory systems infeasible, or where network limitations cause bottlenecks for broadcast-type communications.

Edge-cut partitioning is employed, rather than vertex-cut with replication (as in other chapters), due to the aim of performing exact full-batch GCN computations. Replication has the potential to further improve running time efficiency with the help of caching, but would introduce additional synchronisation communication overheads. In order to maintain the focus on examining exact communication overheads during training and inference, replication is not explored here.

The method is shown to be scalable on a CPU cluster with an MPI backend, with the use of a hypergraph-based partitioning providing significant speedups compared to SOTA alternatives. The latency between hidden layers is considerably amortised, allowing deeper GCN models to be trained, and with no impact on accuracy performance. Experiments on a GPU cluster with NCCL backend also provide useful insights on large-scale GNN training, demonstrating less scalability compared to the CPU versions as well as slower running times in some instances.

This solution utilises a hypergraph-based model for static partitioning of the data in full-batch GNN training, and additionally makes use of a hypergraph sampling trick for mini-batch GNN training. The novel stochastic hypergraph model is designed to successfully capture the randomness of communication operations in parallel mini-batch training and further improve scalability. It uses a sampling and merging technique to construct a hypergraph that encodes the expected (rather than exact) communication volume among processors and thereby achieves improvements over the hypergraph model.

The parallel GCN training algorithm is adaptable to other GNNs by changing only the local computations without requiring any changes in terms of the communication operations. The asynchronous communication scheme is able to significantly alleviate the scalability issues of distributing GNN tasks on massive datasets to a large number of machines. In the next chapter, the partitioning task is considered for a fully streaming case where the subsequent GNN can perform asynchronous updates. While the goal of this chapter was to design a communication-efficient algorithm for systems having memory and network bandwidth limitations, in the upcoming streaming case the target is low latency systems.

Chapter 7

Hypergraph-based Partitioning on Streaming Graphs for GNNs

> *It's weird how I am constantly surprised by the passage of **time** when it's literally the most predictable thing in the universe.*
>
> Randall Munroe, *xkcd (Star Wars)*

The previous chapter dealt with GNNs in the context of static graph inputs. However, social networks [166], item-product recommendation systems [179], physical systems [137], and biological networks [134] are among the many examples of graph data that are inherently dynamic with evolving topology and features over time. Chapter 4 offered a graph analytics approach that could operate on temporal networks. In this chapter, a similar dynamic input, ingested in the form of streaming graph events, is considered for graph learning tasks.

GNN models on streaming graphs must be equipped to continuously capture the dynamic state of the graph and incrementally update the model, while ensuring that system latency, memory, and throughput requirements are met during both inference and training. Streaming GNN systems, such as the recent D3-GNN system [63] built atop Apache Flink, can help to tackle all these challenges. In order to distribute the computations in such streaming GNN systems, it is necessary to manage the computational load balance while minimising the communication costs using a suitable online partitioner, the design of which is the focus of this chapter.

While graph streams typically consist of node/edge addition and feature update events to the graph, node/edge deletion events require careful handling within the GNN system to ensure that the 'staleness' problem is avoided in the resulting representations, which is an orthogonal area of work. The partitioning may also become imbalanced after numerous deletion events, and require re-balancing. In this chapter, the partitioner is designed under the assumption of a stream having additive nature only (without any deletion events).

For efficient distributed processing of graph streams in low latency environments, it is of utmost importance to develop a high quality streaming partitioner that does not cause bottlenecks in the system. This chapter presents the Partitioner component developed for the D3-GNN system, which is designed to be data-parallel by assigning parts to graph elements on-the-fly using streaming graph partitioning heuristics. To ensure that the throughput of the system remains unimpeded, the Partitioner is designed in a multi-threaded manner that grants some latency optimisations. Such a partitioner must also help the system to tackle GNN challenges such as 'neighbourhood explosion' caused by the MPGNN aggregation step [11]. To this end, a flexible mapping of logical parts to physical parts is proposed that allows the system to scale the different GNN layers to a variable number of processors.

While streaming graph inputs already present a significant technical challenge, more complex streaming hypergraph data structures may also need to be ingested. Such hyperedge streams can be viewed as a representation of traditional vertex streams but having incomplete adjacency information, which are not supported by any popular dynamic GNN systems. Thus, the streaming partitioning framework in this chapter is developed to support distributed, streaming graph learning tasks on streaming hypergraphs as well as streaming graphs. The research into streaming hypergraph partitioners is limited, and this chapter presents the first experimental comparison of graph- and hypergraph-based techniques for streaming graph learning workloads. To compare the performance of hypergraph neural networks (HGNNs) with GNNs, and thereby judge the efficacy of partitioning hypergraph streams over graph streams for distributing the respective models, hypergraph transformation techniques are applied to generate equivalent graph approximations.

Unlike the edge-cut partitioners in the previous chapter, the streaming partitioners designed here are vertex-cut. Hence, similar to Chapter 5, here the partitioning involves vertex replication. This is due to streams of edges/nets being processed, where assigning them to parts can create cut vertices that must be replicated onto multiple parts. Distributed computation with replicated partitioning introduces new challenges that must be handled, such as maintenance of master part tables that the replicas will utilise for communication, and decisions on which replicas require active access to feature values; these choices vary depending on whether the input is treated as a graph stream or a hypergraph stream. The proposed framework enjoys scalability to high degree of parallelism, up to 200 CPU cores, via distribution of tasks by the streaming partitioning solution. It is applicable on large-scale dynamic GNN workloads that can be in the form of either graph streams or more complex, and previously unstudied, hypergraph streams.

7.1 Preliminaries

Streaming dataflow system

A streaming dataflow system is a pipeline of data transformation tasks, also referred to as 'operators', that consume input streams and emit output streams. The pipeline starts at one or more 'data-source' operators and ends at one or more 'data-sink' operators. Operators are parallelisable across threads and machines, with each parallel 'sub-operator' instance performing local computations.

Apache Flink [25] is a versatile, stateful dataflow framework that can process both bounded and unbounded streams, with checkpoints for failure recovery. Flink has the ability to trigger computations based on event timestamps and watermarks for consistency and completeness, and based on processing time for latency requirements.

Hypergraph star expansion

The star expansion algorithm for a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ can be used to construct a heterogeneous bipartite graph $\mathcal{G}^* = (\mathcal{V}^*, \mathcal{E}^*)$. A new vertex is created for every net $n \in \mathcal{N}$, resulting in a new vertex set $\mathcal{V}^* = \mathcal{V} \cup \mathcal{N}$. Each such vertex $n \in \mathcal{V}^*$ is connected to all the vertices $v \in \mathcal{V}^*$ that were contained in n as a hyperedge of $\mathcal{H}$, i.e., all the $\text{pins}(n)$. Therefore, each net corresponds to a star in the graph $\mathcal{G}^*$, and the new edge set is given as $\mathcal{E}^* = \{(v, n) : v \in n \text{ and } n \in \mathcal{N}\}$.

In the distributed streaming setting, performing star expansion is useful as it generates an edge stream. This allows for vertex-cut replication-based partitioning approaches to be used. By contrast, clique expansion generates a vertex stream which is better suited to edge-cut partitioning without any replication.

7.2 Streaming partitioner design

The D3-GNN system [63] is designed to perform asynchronous forward and backward pass operations optimised for online inference and incremental training of GNNs on streaming graph data. The asynchronous computations are distributed with the help of streaming graph/hypergraph partitioning. This is performed by a Partitioner operator that is designed to be embedded within its dataflow pipeline of operators (seen in Figure 7.1).

GNN embedding layers in the system are defined via the behaviour of the MESSAGE (γ), AGGREGATOR (α), and UPDATE (ψ) functions following the common MPGNN framework. These layer computations (i.e., Storage operators) are stacked to form the overall dataflow pipeline. *Model-parallelism* is achieved across the Storage operators because each layer of the GNN is processed by a separate operator. Meanwhile, *data-parallelism* is enacted across their sub-operators that process smaller subgraphs, enlisting the Partitioner to divide the incoming graph/hypergraph stream into parts. This section describes the partitioning scheme designed to support the distributed execution of the hybrid-parallel pipeline.

Replication for inter-partition communication

Streaming partitioning logic is needed to assign parts to the incoming graph elements using a local dynamic summary of the graph seen so far. Any streaming partitioning algorithm can be embedded within the Partitioner operator. For example, this could rely on an HDRF partitioning algorithm [130] for edge streams, or a streaming min-max hypergraph partitioning algorithm [4] for net streams. Apart from partitioning the graph, the Partitioner operator also routes the workload to the relevant logical parts. At any time that a feature update must take place, it is the job of the Partitioner to identify the master part of its corresponding graph element and assign it to this task.

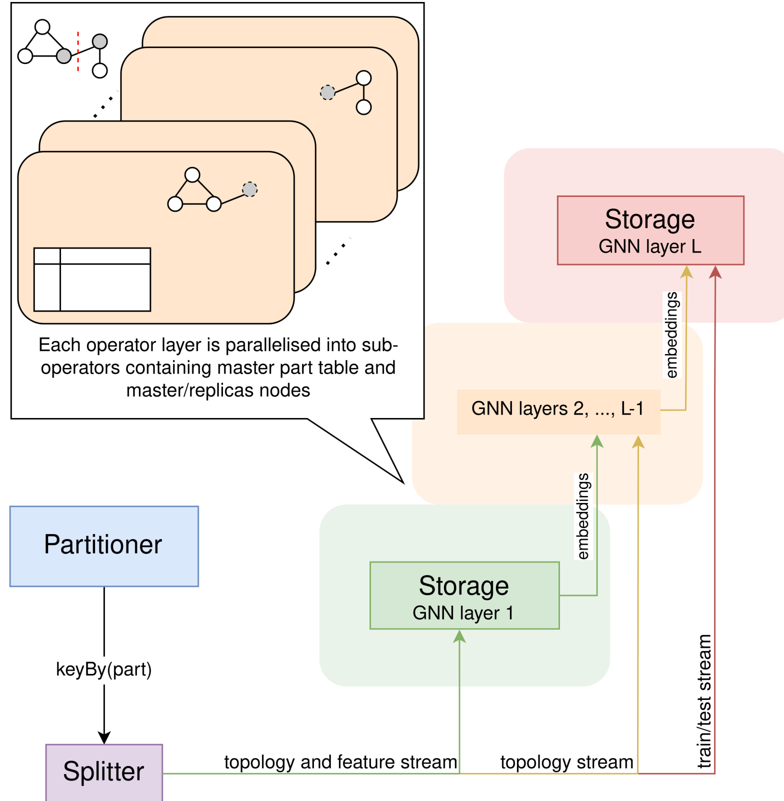


Figure 7.1: Streaming partitioner functionality within the dataflow pipeline

When the data is in the form of an edge stream (e.g., user-user interactions in social networks), the HDRF vertex-cut algorithm is adapted for use within this system. In other words, the stream of edges (u, v) being ingested gets assigned to different partitions on-the-fly, resulting in some cut high-degree vertices that belong in multiple parts. The first assigned part is treated as the master part and subsequent ones as replica parts. In this way, through the master-replica tethers recorded by the Partitioner, communication is possible between Storage operators that maintain only their local partitioned subgraphs after replicating the cut vertices. For the vertices that are replicated, their corresponding computation units (AGGREGATORS) reside only with the master vertex (and not with replicas). GNN aggregations that occur here instigate cascading feature updates in the node neighbourhood, which is accessed in full through various replicas located in other parts as well. These update operations are allocated to the appropriate logical parts by the Partitioner.

On the other hand, when hypergraphs are streamed in (e.g., user and list of topics that they interact with in an online community), the streaming min-max algorithm is adapted for use. Here the incoming vertices are assigned parts such that there is maximum intersection between its nets and the nets already connecting to the part, i.e., to minimise the greatest number of distinct nets being connected to any part. Therefore, instead of replicated vertices, this leads to replicated net elements across different parts that must communicate with one another. These nets have their corresponding computation units (AGGREGATORS) replicated too, as they are required during updates to participate in the forward and backward propagation steps with local partial matrices. This is because an HGNN is treated as having a

two-level message passing and aggregation process, one within every hyperedge (intra-edge) and another across the hyperedge (inter-edge). The latter of these requires aggregation at the nets (a more detailed description can be found in [7]).

Algorithm 7.1: Streaming Partitioner

input: $state, master_table, num_parts, operator$

1 *part* ← *assignPart(state, master_table, num_parts, operator)*
2 **if** $master_table(operator.element) = \emptyset$ **then**
3 | $master_table(operator.element).insert(part)$
4 $assignMaster(operator.element, master_table)$
5 $operator.part \leftarrow part$
6 **return** $operator$

Algorithm 7.1 describes the pseudo-code for this streaming partitioner logic. The total number of available parts is given by num_parts, and the current summary of the dynamic graph is in $state$. The chosen partitioning method assigns a $part$ number (line 1), which is declared as the master if the $element$ has not been seen previously in the stream and therefore does not have any pre-existing master part recorded (lines 2–3). The $element$ stores its assigned $part$ (be it master or replica), as well as information on how to access the master (lines 4–5).

In order to communicate with their masters, replicas store their master part number. Masters, in turn, maintain a list of their replicated parts. To eliminate storage redundancy, some elements are not actively replicated, and instead only communicate via empty stubs in the replicas. For instance, vertex AGGREGATORS are stored only in the master and need to merely receive messages from replicas. For such use-cases, replicas do not need to access the features stored in their masters, but instead the replicas and master merely need to know the existence of each other for communication. AGGREGATORS for nets, by contrast, must be stored in both the master and replicas as they update all their respective local part vertex features (during the inter-edge step).

Combating neighbourhood explosion

Changes to the data in Storage, brought about by graph feature updates, must induce corresponding changes in AGGREGATORS and cascading MESSAGES for maintaining up-to-date representations. Due to this cascade, with every subsequent layer, the amount of computations necessary in the GNN forward pass increases. The increase is roughly by a factor that is the average node degree of the input graph. Since the aggregation computation also first involves message passing steps, this neighbourhood explosion problem triggers additional communication costs in a distributed system.

The streaming setting can face bottlenecks due to the neighbourhood explosion problem in GNNs, which can cause Storage operators for deeper layers to receive higher forward pass workload. This imbalance in workload patterns within different nodes in the GNN may be aggravated by hubness (centrality) of certain nodes or by external factors that localise the updates topologically.

To tackle the neighbourhood explosion challenge, a new system hyper-parameter is introduced called 'explosion factor' (ω). This enables the parallelism obtained from the Partitioner to be varied for each individual Storage operator, i.e., the number of sub-operators that perform the same task in a data-parallel manner now depends on the GNN layer. Namely, given an initial parallelism p and L layers of the GNN, the actual parallelism for each Storage operator (layer) is assigned as $p_i = p * \omega^{i-1}$ for $i \in [1, \ldots, L]$. This parameter must be selected considering the frequency of training, even though the forward pass is always benefited by higher ω, because neighbourhood explosion has a reverse effect on layer-wise workload during the backward pass for training.

Variable parallelism using logical parts

Assigning physical partitions alone does not allow flexible re-scaling of Storage operators (e.g., if the number of physical partitions changes due to failure), nor does it support different parallelisms across the chained Storage operators (e.g., to better cope with the exponential load induced by neighbourhood explosion). For this, the total number of available parts (num_parts used in Algorithm 7.1) is defined to be the same as the maximum possible parallelism of the system (p_{max}), while graph elements are actually partitioned using $keyBy(operator.part)$. In other words, the streaming Partitioner assigns only *logical parts* while the *physical part* is computed using a hash of the assigned logical part. As a consequence, multiple logical parts may map to the same sub-operator.

When an operator's parallelism p gets closer to p_{max}, some sub-operators may remain constantly idle due to never being assigned with any logical parts. To circumvent this, instead of using Flink's default hashing method, Algorithm 7.2 is employed to compute physical parts from logical parts. Each operator gets assigned at least one key, and overall, logical parts are evenly distributed to operators depending on the current p.

Algorithm 7.2: Physical Part Computation

input: $logical_part$, p, p_{max}

1 $key_group \leftarrow logical_part \;\%\; p_{max}$

2 $physical_part \leftarrow key_group * p / p_{max}$

3 **return** $physical_part$

Operators store data (state) in two ways: Operator State stores data for a given sub-operator, which can be accessed by all elements arriving at sub-operator, while Keyed State stores data at the granularity of a unique key, and each arriving element can only access data that is assigned to its particular key. Upon re-scaling the system, Operator State is either randomly redistributed to new sub-operators, or broadcast (in entirety) to all remaining sub-operators to then perform recovery logic. Keyed State, however, is distributed to the new sub-operator containing that key. This flexible mapping of keys to sub-operators allows for re-scaling of the physical partitions based on availability, and a fixed hash function (for logical to physical parts) guarantees fault tolerant recovery. Hence, the fault tolerance logic can be delegated to Flink, thereby ensuring state redistribution and correct operation even under variable parallelisms.

Latency optimisation

Since the streaming graph input is in the form of an edge stream, an online partitioning step followed by the incremental forward pass update step takes place for every new edge event that is ingested. In the hypergraph stream case, the partitioner assigns a single vertex to its part but the downstream HGNN carries out a number of updates (both intra-edge and inter-edge) corresponding to this one partitioning step. Therefore, it is observed that the partitioner can cause a bottleneck in the system when a graph stream is ingested, affecting the latency of the subsequent GNN inference operation. This is less of a concern in the hypergraph stream where the partitioner can more easily keep up with the throughput of the inference forward pass. To overcome the bottleneck, the Partitioner operator is designed with a novel distributed setup to allow for latency optimisation wherever necessary (e.g., when using HDRF with edge stream).

Distributing the HDRF partitioner within D3-GNN requires a shared-memory model for storing partial degree and partition tables, something currently unavailable in Apache Flink. Without it, a single thread needs to be allocated to the partitioner, which causes the aforementioned bottleneck when scaling up the system. Hence, the Partitioner operator is developed to support correct, concurrent thread distribution for any streaming partitioner methods adapted within it for use in the system. It distributes the main partitioning logic among an arbitrary number of threads while having synchronised access to the output channel. The latter is necessary to avoid corrupt data during network transfer, as output channels consume data in smaller units than the graph data being streamed. Furthermore, a vertex-locking mechanism is proposed for correctness within HDRF. That is, edges with common vertices are assigned to their logical parts one at a time. Note that for shuffled graph streams (streams not generated via BFS or DFS traversals), non-locking partitioning is not found to significantly impede the partitioning quality (load balance and replication factor).

7.3 Experimental evaluation

The performance of D3-GNN is evaluated in the distributed setting using the proposed Partitioner logic. Both graph and hypergraph streams are considered as input dynamic workloads, with subsequent inference computations done using GNNs and HGNNs.

Datasets

The Partitioner in the D3-GNN system is evaluated on two hypergraph datasets (**`tags-ask-ubuntu`** and **`threads-math-sx`**) that are publicly available[1] as timestamped sequences of simplices (nets) [16], as well as on the star expanded graphs that are generated from these hypergraphs. In `tags-ask-ubuntu`, nodes denote tags and their corresponding nets represent questions on the Ask Ubuntu forum. A node (tag) can thus be associated with a list of nets (questions) that have the same tag. In `threads-math-sx`, nodes are users and nets represent message threads on the Math Stack Exchange forum. A node (user) has a list of nets (threads) that the user has participated in.

[1] https://www.cs.cornell.edu/~arb/data/

To examine the performance of the Partitioner more closely on streaming graphs, a temporal network dataset **reddit-hyperlink** [97] is used directly, as obtained from the Stanford Large Network Dataset Collection.[2] reddit-hyperlink dataset contains an edge-list of directed subreddit mentions (hyperlinks) derived from the Reddit Social Network. The hyperlink data is provided in two files based on mentions in either the title of the post or in the body, but only data from the body is considered in this chapter. The data is processed in sequence and treated as an incoming stream of edge addition and feature update events to the graph, ordered by the edge timestamps.

The properties of these datasets are listed in Table 7.1. The number of edges of the hypergraph datasets is based on their star expanded graph.

Table 7.1: Dataset properties

Dataset	#Nodes	#Edges	#Nets	Type
tags-ask-ubuntu	3,029	1,468,584	271,233	Directed
threads-math-sx	176,445	3,220,786	719,792	Directed
reddit-hyperlink	35,776	286,561	-	Directed

Baselines

For experiments on graph data, a **GNN** model is implemented in the D3-GNN dataflow pipeline, using a distributed 2-layer GraphSAGE architecture [67]. This model is also run on the star expanded graphs obtained after transforming the hypergraph data. For operating directly on hypergraphs, the **HGNN** model is a 2-layer HyperSAGE architecture [7]. Both models use a *Mean* aggregator. The focus in this chapter is on comparing the GNN and HGNN performances with graph and hypergraph stream inputs within the D3-GNN system, therefore comparisons against other systems are omitted.

A more nuanced comparison of the Partitioner performance is also provided, for which **HDRF** partitioner and **MinMax** (streaming min-max hypergraph partitioner) are used for graph edge streams and hypergraph vertex streams respectively. For distributing graph tasks, the custom HDRF method is applied with balance coefficient $\vartheta = 8$ unless otherwise specified. For hypergraphs, the MinMax algorithm is adapted keeping the default balance tolerance as defined [151]. A **Random** partitioner baseline is also employed.

Setup

Experiments are executed on a Slurm cluster with 10 machines, where each machine contains Xeon E5-2660 v3 @ 2.6 GHz (20 cores/40 threads) and 64GB RAM. Apache Flink[3] and Deep Java Library[4] with PyTorch[5] are used as the primary ML framework.

The explosion factor parameter for distributed task allocations (to tackle variable workloads at each layer due to neighbourhood explosion in GNNs) is $\omega = 3$ unless specified. That is, when the parallelism of the system is said to be p, the parallelism for each layer is

[2]https://snap.stanford.edu/data/
[3]https://flink.apache.org
[4]https://djl.ai
[5]https://pytorch.org

$p_i = p * 3^{i-1}$ for $i \in [1, \ldots, L]$ as described earlier. Depending on Flink's internal allocation during execution, this translates to distributed tasks using $\frac{p}{10}$ machines on average.

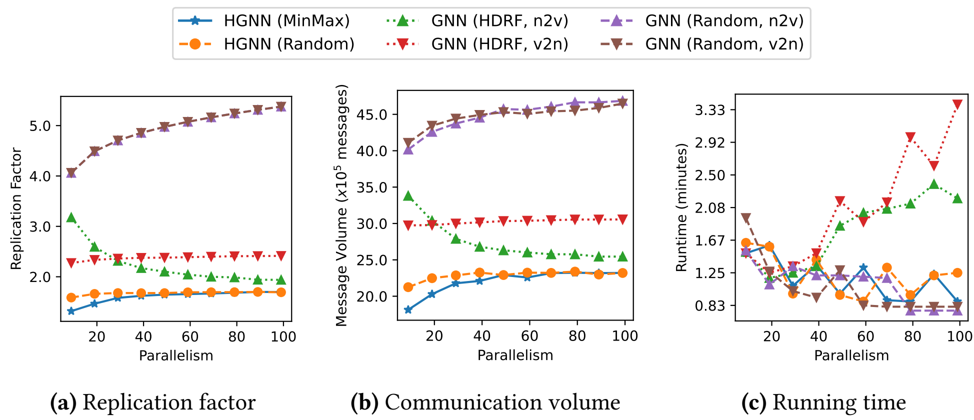
Results are reported for an average over 3 runs, with the systems being scaled up from 1 machine (20 cores) to 10 machines (200 cores). For these experiments the number of logical parts in D3-GNN is set as the parallelism of final layer (i.e., ωp for the 2-layer GNN).

Comparison of HGNN with GNN

First, the performance of the partitioners is compared in terms of the following metrics: i) replication factor in the partitioned graph/hypergraph to reflect how many replica vertices/nets are generated, ii) communication volume in the first layer of the GNN or HGNN to study the overheads, and iii) running time of the GNN or HGNN model to evaluate efficiency performance gains.

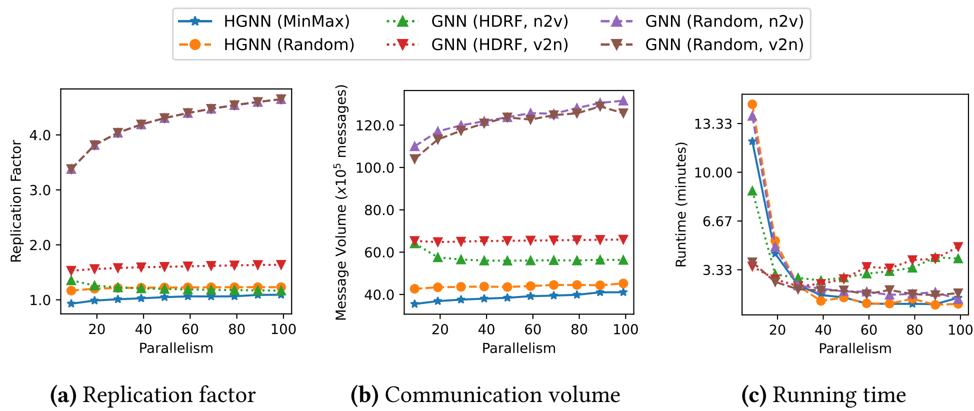
The HGNN 2-layer HyperSAGE model is compared against the GNN 2-layer GraphSAGE model. The two hypergraph datasets are used as streaming input directly by the HGNN, which performs a two-level aggregation as described earlier. The distributed HGNN model is made data-parallel with the help of a MinMax partitioning method to place the incoming vertices. The same hypergraph datasets are transformed into their equivalent graph format by means of the hypergraph star expansion method. These star expanded graphs are then used as inputs for the GNN model. The GNN model uses the HDRF partitioning method to distribute the edges to parts for data-parallelism. In this manner, a comparison is obtained between the graph and hypergraph partitioning approaches in a streaming graph learning setting, which has not been previously studied.

As can be seen from Figures 7.2 and 7.3, using an HGNN model to directly operate on hypergraph data is preferable to using a GNN on transformed data (star expanded graph). This echoes the findings of Arya et al. [7] for the star expansion case, with a new focus on streaming inputs. Hypergraphs are capable of encoding more information that may be lost during the transformation into graphs. Given a streaming system with comparable latency and throughput for both models, an HGNN would be the superior choice over a GNN. That being said, the two-level aggregation in an HGNN model engenders more communication operations. Consequently, it is necessary for the hypergraph partitioning for distributed HGNN to produce much better data locality than the graph partitioning for the equivalent distributed GNN. From the figures, it is evident that the replication factor directly influences the network overheads, as the message volume follows similar trends. Both are seen to be significantly lower for the HGNN, and scalable to high parallelism on both `tags-ask-ubuntu` (Figures 7.2a and 7.2b) as well as `threads-math-sx` (Figures 7.3a and 7.3b). That is, even despite the greater number of communications needed by HGNN, the overheads in the D3-GNN system are maintained lower than those of the GNN version. Running time efficiency improvements of HGNN over GNN are even more pronounced on the larger and sparser `threads-math-sx` graph, which can be explained due to hypergraph partitioning being particularly powerful for sparse matrix operations (Figures 7.2c and 7.3c).



(a) Replication factor **(b)** Communication volume **(c)** Running time

Figure 7.2: Performance comparison with GNN and HGNN on graph and hypergraph formats of `tags-ask-ubuntu`



(a) Replication factor **(b)** Communication volume **(c)** Running time

Figure 7.3: Performance comparison with GNN and HGNN on graph and hypergraph formats of `threads-math-sx`

Effect of star expansion

The above findings suggest that star expansion may not be suitable in a low latency distributed setting for graph learning. Moreover, star expansion is primarily designed for static cases. Given a hypergraph stream, the conversion into a star expanded graph stream is found to induce an inherent grouping in the resultant overall stream. That is, given a vertex v and its list of nets $\{n_i\}$ (i.e., a single input event in the hypergraph stream), the star expanded graph produces a stream of undirected edges $\{(v, n_i)\}$ all with v as one endpoint (i.e., a sequence of inputs in the graph stream). To study the impact of this ordering, the star expansion process is also reversed to generate $\{(n, v_j)\}$ sequence of edges instead, where all the vertices connected to the same net are grouped in the overall stream rather than vice versa. These two versions of the resulting graph are referred to as 'v2n' and 'n2v' respectively to denote whether the edge stream is grouped as 'vertex to all its nets' or 'net to all its vertex pins'.

Using 'n2v' has a significant detrimental effect on partitioning quality of HDRF and therefore message volume, and also severely impacts the running time of the GNN on both datasets (Figures 7.2 and 7.3). This suggests that supplying a stream of edges connected to the same source vertex, before moving on to the next vertex, is a more suitable transformation in the

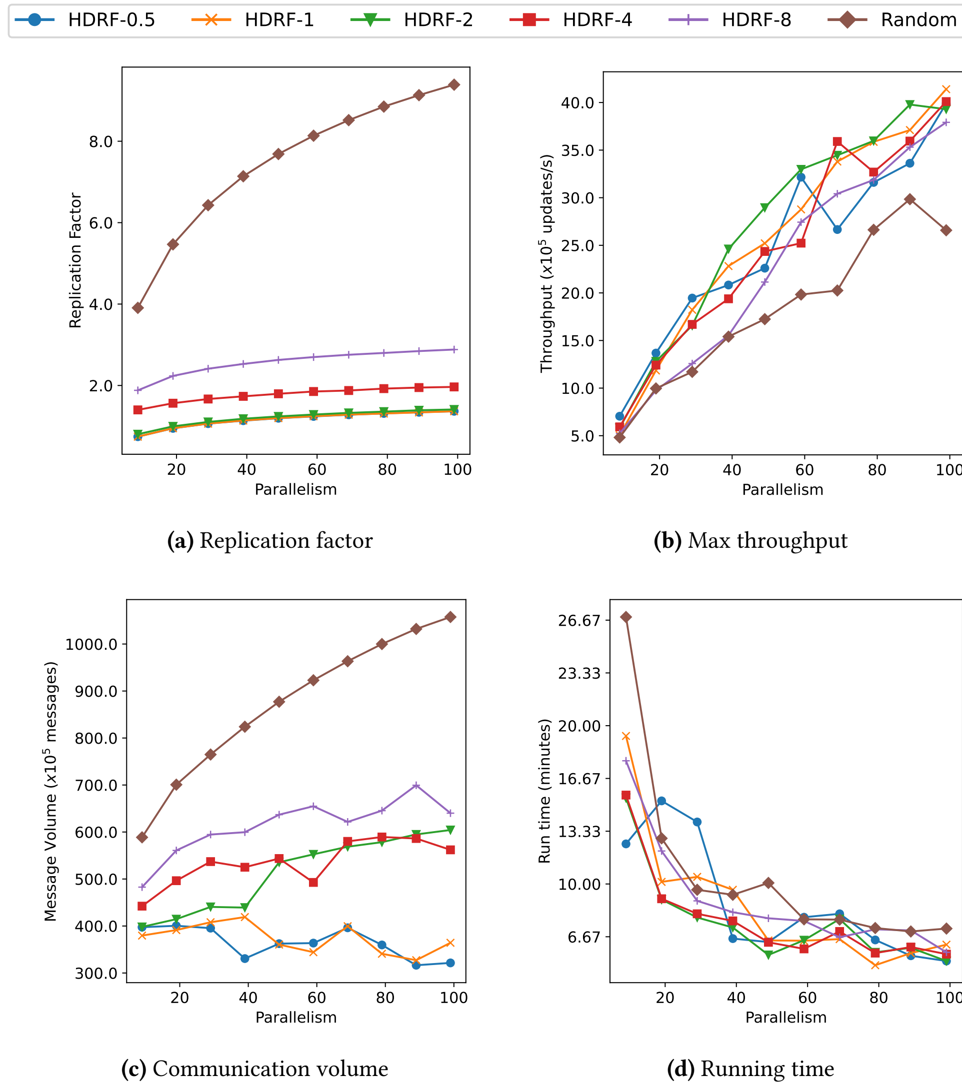


Figure 7.4: Performance comparison with different partitioners on `reddit-hyperlink`

streaming setting. The 'n2v' version is somewhat closer in structure to a streamed clique expansion, where all vertices having a net in common get connected as a clique. Based on these findings, the clique expansion, which is more commonly applied to transform hypergraph inputs for HGNNs that require graph data, may not be the ideal technique.

Effect of partitioner

Figures 7.2 and 7.3 show that MinMax partitioner for the hypergraph stream is advantageous over other configurations in terms of message volume and running time efficiency. The Random partitioner performs reasonably well on the hypergraph for the small and dense `tags-ask-ubuntu` dataset but incurs much more network overheads with HGNN on larger datasets. For the GNN model, Random edge allocation enjoys short distributed running times, but the HDRF partitioner outperforms Random when considering communication costs. Regardless, even HDRF for distributed GNN has significantly poorer running time efficiency as compared to any partitioned HGNN model. This once again speaks to the need for

supporting direct computations on hypergraphs instead of relying on graph approximations.

The MinMax partitioner assigns an incoming vertex and all its nets to their allocated partitions in a single step, with the imbalance tolerance value being based on the average part weight at that point in the streaming partitioning process. HDRF, on the other hand, assigns edges to parts and uses a predefined imbalance tolerance value ϑ. Therefore, a further study is conducted on HDRF by varying its ϑ parameter. A performance comparison within the D3-GNN system between HDRF with different parameter settings (ϑ) along with a Random partitioner is shown in Figure 7.4. For each ϑ, the corresponding label in the figure is 'HDRF-ϑ'. This imbalance tolerance at higher values assigns more importance to the load balancing constraint. Lower ϑ improves network communication overheads at the cost of load balance, mainly due to a lower replication factor, but can find uses to meet bandwidth limitations of the distributed environment. Using Random partitioning results in the highest communication volume needed for the GNN, thereby also lowering its throughput.

7.4 Discussion

This chapter explores partitioning in a streaming setting; graph data is ingested in the form of a stream of edges, and complex hypergraph data is ingested as a stream of nets. Hypergraphs are equivalent, in their dual transformation, to bipartite graphs with nets being represented as a new type of vertex. Hence, with the help of a star expansion technique, the streaming hypergraphs may be unrolled into edge streams, thus allowing comparisons between graph learning models that operate on these two different formats. Streams are considered to only add nodes/edges to the graph or update their features, with no deletion events.

Here, partitioning tactics akin to those from Chapter 6 are used to create a data-parallel pipeline. Ideas related to vertex replication, similar to Chapter 5, are deployed in tandem with this setup, since the streaming edges/nets are to be distributed using a vertex-cut partitioning scheme. This requires replication logic to be incorporated within the partitioners to allow masters and replicas to communicate. Distributing such an incremental system also calls for the partitioner to support variable parallelism to combat neighbourhood explosion in deeper layers, and the ability to flexibly re-scale the parts for fault tolerance. For the partitioner to keep pace with the low latency pipeline on streaming data, multi-threaded latency optimisations are also incorporated.

The HGNN model operating directly on the hypergraph is shown to be more suitable, compared to a GNN on the corresponding transformed graph, in maintaining efficiency performance and reducing communications. Distributing the hypergraph via MinMax partitioning and using a data-parallel HGNN is preferable to distributing the edge stream with HDRF and using a data-parallel GNN, especially with large, sparse inputs. The grouping of edges in the star expansion is also found to impact the partitioning quality of HDRF.

Different partitioners are examined in more detail for the GNN workload, by varying the imbalance tolerance of HDRF for graph stream data. This reveals a tradeoff between load balance and replication factor, with the latter contributing to higher communication costs during inference and training. Nonetheless, the flexible parallelism of the system, achieved

using the explosion factor parameter, supports its high scalability to large number of processors. Both the throughput and running time of the model are found to improve with higher parallelism. Therefore the use of appropriate partitioners helps to distribute the streaming D3-GNN system easily for both hypergraph and graph data.

Chapter 8

Conclusions

> “ *Too many scholars think of research as purely a cerebral pursuit. If we do nothing with the knowledge we gain, then we have wasted our study. Books can store information better than we can–what we do that books cannot is interpret. So if one is not going to draw* ***conclusions****, then one might as well just leave the information in the texts.* ”
>
> Brandon Sanderson, *The Way of Kings*

To conclude this thesis, a discussion of all the solutions is presented to review and highlight their main contributions, which are also available at a glance in Table 8.1. This is followed by a critical summary of the limitations and avenues for future research.

8.1 Discussion

Several challenging research questions were addressed in this thesis, first to identify the salient features of hypergraphs and dynamic graphs, and then to apply hypergraphs in different use cases, both static and dynamic, to optimise computations in graph analytics and learning tasks.

In particular, Chapters 4 and 5 investigated graph analytics applications. Chapter 4 used hypergraphs to efficiently encode numerous sampled subgraphs and thereby reflect the expected spread of information (from arbitrary seed nodes) through a temporal network. Chapter 5 applied hypergraph (and graph) models for replicated partitioning of a search graph to achieve scalable approximate single-source similarity calculations. Chapters 6 and 7 focused on the graph-based learning domain, more specifically, GNN models. Chapter 6 used hypergraph partitioning to better encode network traffic and described the design of an asynchronous communication scheme for scalable distributed training of static GNNs. Chapter 7 developed a streaming hypergraph (and graph) partitioner that interfaced with an

Table 8.1: Summary of thesis contributions

Challenges	Proposed Novel Solutions	Findings	Applications
Chapter 4: Graph analytics (spread modelling) on dynamic graphs	T-IC, a temporal independent cascade model that uses hypergraph-based sampling to generate reachable sets where temporal dependencies are taken into account (based on dynamic graph topology and dynamic propagation patterns) while maintaining solution quality guarantees; RSM and ESM algorithms to find efficient solution sets for the identification of sentinel and susceptible nodes respectively	RSM and ESM outperform 5 baselines to identify solution sets on dynamic graphs; T-IC model remains feasible on large-scale graphs unlike baselines; Both temporal and global patterns are taken into account by hypergraph sampling and fully customisable spread characteristics are supported	Monitoring/combating the evolving spread of diseases, computer viruses, misinformation
Chapter 5: Graph analytics (similarity search) on large-scale static graphs	Hypergraph- (and graph-) based partitioners applied along with vertex replication and caching techniques for scalable approximate RoleSim* search on the pruned subgraphs obtained from the large-scale original graph; Evaluation of SSRS* solution quality based on successful identification of duplicated nodes, when partitioning is done with and without strict data locality, to appreciate differing task-specific workload patterns	Caching replicated vertices improves SSRS* computation efficiency; Strict data locality involves accuracy tradeoff; Hypergraph partitioner lowers running time by 15% over graph partitioner	Scalable similarity search for recommendation systems, web search, document retrieval
Chapter 6: Distributed GNNs on static graphs in limited memory/bandwidth environments	Hypergraph-based partitioning with non-blocking point-to-point communication scheme to minimise communication overheads for distributed data-parallel full-batch GNN training; Hypergraph construction approach using stochastic sampling to further optimise distributed mini-batch GNN training	Hypergraph partitioning reduces communication overheads in full-batch GNN training with 10x speedup over DGL baseline; Hypergraph sampling can encode randomness of mini-batching for further improvements	Highly scalable graph-based learning tasks such as question-answering, protein matching, traffic forecasting
Chapter 7: Distributed GNNs on streaming graphs in low latency environments	Hypergraph- (and graph-) based partitioning along with vertex replication to support distributed online inference and incremental training of HGNNs and GNNs on streaming graphs; Transformation between complex hypergraph streams and graph streams with support for incomplete adjacency information in dynamic systems	HGNNs on hypergraphs are more efficient than GNNs on star expanded graphs especially for large, sparse graphs; MinMax partitioner distributes hypergraphs to incur lower network overheads compared to HDRF partitioner on edge streams	Low latency streaming graph-based learning for traffic forecasting, social networks, recommendation systems

asynchronous dataflow system, and took advantage of replication to perform online GNN inference and training in low latency settings.

The Temporal Independent Cascade (T-IC) model presented in Chapter 4 provides an IC-based estimation of the set of reachable nodes that can be used even when spread takes place in a network having evolving topology and propagation rates. The defined spread function under the T-IC model yields solutions with quality guarantees due to submodularity even in this dynamic setting. Besides this, two optimisation objectives are put forward; the Reverse Spread Maximisation (RSM) and Expected Spread Maximisation (ESM) algorithms are devised to deliver effective solution sets for these objectives of identifying sentinel nodes and susceptible nodes respectively. Hypergraphs are employed to efficiently encode the sampled subgraphs for the expected set of of reachable nodes during a T-IC process on the network. There are two levels of randomness in this sampling strategy, which imparts new capabilities: i) modelling temporal dependencies during the spread process, and ii) estimating the potential spread from any random seed node. Thus, hypergraphs can offer innovative strategies to furnish dynamic graph tasks with a temporally-guided high-quality stochastic sampling methodology.

Chapter 5 makes use of hypergraph-based (and graph-based) partitioning on a similarity search graph. Here, a pruning strategy is put forward that uses cached values from the replicated vertices instead of recomputing similarity scores. The method is shown empirically to produce results with an efficiency tradeoff that depends on whether or not inter-partition communications are allowed. The triangle inequality property of the similarity measure is used to guarantee correctness of the similarity computations in the pruned subgraph. Therefore, exact similarity computation is also a possibility that is discussed. Furthermore, this chapter elicits a deliberation about the variations in workload patterns when dealing with subtly different graph analytics tasks. This discussion, for different similarity measures, hinges upon an unsupervised evaluation setting that expects (sampled neighbourhood) duplicates of a node to be ranked highly in the node's top-k similarity list. These differing workloads patterns are all the more significant when analytics tasks are compared against graph-based learning tasks. Nonetheless, hypergraph models are deemed powerful for use within partitioning tools to enable distributed computations for both analytics and learning on graphs, and caching the states of replicated vertices proffers some interesting avenues for performance speedups.

The hypergraph model for partitioning is then utilised to good effect in designing the novel asynchronous point-to-point communication scheme described in Chapter 6. Using an edge-cut hypergraph partitioner helps to better encode the communication volumes during irregular data accesses induced by the neighbourhood aggregation step of graph convolution operations. This holds true especially for graphs with sparse adjacency matrices, since sparse matrix multiplications are a core kernel operation in GNNs. The developed parallel algorithm is helpful in minimising overheads when GNNs are scaled to a large number of processors for distributed full-batch training on massive graphs, with no adverse impact on accuracy performance. A stochastic sampling approach together with the hypergraph model provides further improvements in the mini-batch training case.

Finally, Chapter 7 takes the notion of asynchronous computations to an extreme, for fully streaming graph workloads handled within a low latency dataflow framework. Here, streaming hypergraph (and graph) partitioners are designed that can combat the neighbourhood explosion problem brought about by the aggregation step in GNNs. This is accomplished using novel variable parallelism at every GNN layer, which also aids in fault tolerance to recover from failure. Additionally, vertex-cut partitioning is employed with intelligent vertex replication to determine where aggregated features should be cached for maximum efficiency. In latency critical applications, even the streaming partitioner itself is distributed to keep pace with the streaming GNN computations and avoid creating a throughput bottleneck in the dataflow system. This entails vertex-locking to ensure correctness of results despite the concurrent threads of the partitioner. More intriguingly, the system is built to support both graph streams as well as more complex hypergraph streams. With hypergraphs ingested directly, and an appropriate streaming hypergraph partitioning method deployed, this enables the system to perform asynchronous distributed computations even on incomplete adjacency information, which is unsupported in state-of-the-art systems.

8.2 Limitations and future work

While this thesis presents significant novelties towards scalable and efficient systems with static and dynamic applications, the study is not without limitations. In particular, the lack of availability of appropriate datasets and baselines makes it difficult to provide conclusive comparisons of the performance benefits in some proposed methods. Besides this, several enticing extensions are deemed possible that can further the functionality and performance of the proposed solutions.

Availability of datasets

In the T-IC model (Chapter 4), the estimated spread should ideally be compared against real-world ground truth information, e.g., highly granular contact tracing data about the spread of a disease, to test the efficacy of the proposed solutions. However, such data that is both granular and temporal is generally not publicly available (due to valid concerns around privacy). To circumvent this, the evaluation is conducted by simulating several instances of spread and considering the average performance for many such reachable set scenarios. The T-IC model along with the RSM and ESM solutions can be applied to new datasets, therefore future work should incorporate evaluations on any such data that becomes available in various domains.

In the streaming setting, due to the relative nascency of such studies, there is currently limited availability of hypergraph-structured data[1]. As hypergraph streams become more popular, it should be possible to incorporate more heterogeneous features and meta-structures into the existing system (D3-GNN), and handle them using its partitioning component which is presented in Chapter 7.

[1] `https://www.cs.cornell.edu/~arb/data/` is a notably comprehensive and well-documented source at present

Suitability of baselines

Due to the approach of T-IC in which dynamic propagation rates are enlisted, it is quite unlike previously existing approaches that can operate on temporal networks. Most of the baselines are also not scalable, meaning that there was a restricted comparison of the efficiency benefits of the algorithms. Moreover, the objectives of identifying sentinels and susceptible nodes presented in Chapter 4, while often studied in theoretical work and static settings, are not well-examined for temporal networks. As more solutions emerge in future, there is ample scope for new evaluations to compare them against the proposed T-IC algorithms.

The new communication scheme in Chapter 6 for distributed GNNs is primarily designed for full-batch training on CPU clusters. Since most distributed GNN systems draw on sampling techniques to heed memory requirements and attain performance benefits, they are not directly comparable baselines. CAGNET is the only competing method that similarly focuses on full-batch communication optimisations for CPU clusters. The experiments conducted using the GPU version provide reasonable comparisons. All the same, there is potential to extend the proposed solution and make it more easily comparable against some existing baselines, as discussed later.

A similar problem is encountered when evaluating the utility of the partitioner in Chapter 7. Currently, no other existing system boasts the same streaming GNN capabilities, which makes exact comparison impossible. Moreover, HGNNs that perform learning directly on hypergraphs are rare, thus the evaluation instead uses star expansion to apply a GNN on the reduced graph structure as a baseline.

Extensions

Distributed T-IC model: Reachable sets are manifested in the T-IC model in Chapter 4 by constructing a hypergraph's nets out of sampled subgraphs that reflect the spread patterns. While already scalable, this task can further be distributed, whereby multiple smaller hypergraphs would emerge out of the partitioned graph. Future research is liable to find appropriate strategies to combine the resulting information and generate good quality solution sets.

Streaming T-IC model: The T-IC model handles dynamic graphs using snapshots with continuous time intervals, but does not support a fully streaming setup. That is, incorporating ideas from Chapter 7 could be a path towards updating the nets of the hypergraph incrementally as new data is ingested, for a streamed sampling approach. This opens the door to modelling spread in a completely online manner.

Temporal Linear Threshold model: Chapter 4 relied on an IC model to simulate the spread, due to its strong ties with influence maximisation problems. It is interesting to contemplate how the objectives and algorithms might need tweaking to fit an LT process to model spread instead. Since nodes within LT models accumulate activation until a threshold is crossed to activate the node, it will have implications on the temporal dependencies that get encoded within the hypergraph-based sampling strategy being proposed in the chapter.

Directed similarity measure: The investigations into scaling the RoleSim* measure (Chapter 5) formed a small part of a larger study, therefore only preliminary evaluations were conducted

on moderately-sized datasets, both directed and undirected. The influence of the directed properties of edges warrants further examination, since it is governed by the deeper working of the similarity measure. It is credible that directed edges would incur different access patterns during the similarity computations, and consequently affect the partitioning quality.

Vertex-cut partitioning for GNN training: It could be useful to integrate a replicated partitioning component in Chapter 6, similar to the methods in other chapters, whereupon some communications may be avoided at important nodes by replicating them. With this vertex-cut approach, despite additional communication overheads caused by the need for synchronisation of replicas, the computations would be reduced by accessing cached representations at the replicated vertices. Using such a replicated parallel algorithm would also enable comparisons with models such as DistGNN [115] that use similar techniques.

Improvements in communication: For the GPU implementation presented in Chapter 6, incorporating future support for asynchronous communication (as in the CPU implementation) may help overcome the limitations of NCCL to overlap communication and computation for better performance gains.

Generalised GNN solutions: GCN, GraphSAGE, and GAT are all special cases that fall under the purview of the MPGNN paradigm, which is an adequate vertex-centric perspective with which to regard GNNs for the purpose of distributing tasks. The proposed parallel GCN training algorithm in Chapter 6 is adaptable to other GNNs by changing only the local computations as described, and this does not require any changes in the fundamental communication operations. This opens up many future directions of research to come up with task-specific and model-specific optimisations, based on how the GNN architecture affects communications, and which asynchronous computations could be overlapped.

Streaming graph deletion events: The streaming setting of Chapter 7 is restricted to streams of an additive nature, i.e., node/edge deletion events are not taken into account. The GNN system can be designed to counter the staleness problems when deletions take place, which is an orthogonal area of study. More importantly, the proposed partitioner can be updated to include an edge routing table, allowing it to support re-balancing of the partitions when substantial changes in the graph topology after deletions begin to cause an imbalance in the load across partitions.

Clique expansion GNN model: In Chapter 7, the star expansion method was employed to transform hypergraphs into graphs for use with GNNs. The streaming setting emphasises the need for careful ordering of the resultant stream after transformation. Star expansion allows a vertex-cut partitioner to distribute the generated edge stream, whereas an edge-cut partitioner could be used on the clique expanded graph. Thus an additional exploration could be to use clique expansion and an edge-cut streaming partitioner such as FENNEL [155], to investigate the impact on the GNN model performance as well as to compare it with the HGNN model on the original hypergraph.

8.3 Concluding remarks

This thesis explores the power of combining hypergraph structures with optimisation techniques such as stochastic sampling and partitioning, for their use within scalable graph systems. The systems under study may be both static and dynamic, and this work makes progress towards supporting large-scale and dynamic graphs, as well as complex hypergraph representations, to enable high-quality, efficient tools for graph analytics and learning.

Bibliography

[1] Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. Streaming graph partitioning: an experimental study. *Proceedings of the VLDB Endowment*, 11(11):1590–1603, 2018.

[2] Amirreza Abdolrashidi and Lakshmish Ramaswamy. Incremental partitioning of large time-evolving graphs. In *2015 IEEE Conference on Collaboration and Internet Computing (CIC)*, pages 19–27. IEEE, 2015.

[3] Sameer Agarwal, Kristin Branson, and Serge Belongie. Higher order learning with graphs. In *Proceedings of the 23rd International Conference on Machine learning*, pages 17–24, 2006.

[4] Dan Alistarh, Jennifer Iglesias, and Milan Vojnovic. Streaming min-max hypergraph partitioning. *Advances in Neural Information Processing Systems*, 28, 2015.

[5] Chrysovalantis Anastasiou, Constantinos Costa, Panos K Chrysanthis, Cyrus Shahabi, and Demetrios Zeinalipour-Yazti. Astro: Reducing covid-19 exposure through contact prediction and avoidance. *ACM Transactions on Spatial Algorithms and Systems (TSAS)*, 8(2):1–31, 2021.

[6] Ioannis Antonellis, Hector Garcia Molina, and Chi Chao Chang. Simrank++: Query rewriting through link analysis of the click graph. *Proceedings of the VLDB Endowment*, 1(1):408–421, 2008.

[7] Devanshu Arya, Deepak K Gupta, Stevan Rudinac, and Marcel Worring. HyperSAGE: Generalizing inductive representation learning on hypergraphs. *arXiv preprint arXiv:2010.04558*, 2020.

[8] Chen Avin, Marcin Bienkowski, Andreas Loukas, Maciej Pacut, and Stefan Schmid. Dynamic balanced graph partitioning. *SIAM Journal on Discrete Mathematics*, 34(3):1791–1812, 2020.

[9] Ammar Ahmad Awan, Khaled Hamidouche, Akshay Venkatesh, and Dhabaleswar K Panda. Efficient large message broadcast using NCCL and CUDA-aware MPI for deep learning. In *Proceedings of the 23rd European MPI Users' Group Meeting*, pages 15–22, 2016.

[10] Bahman Bahmani, Abdur Chowdhury, and Ashish Goel. Fast incremental and personalized PageRank. *Proceedings of the VLDB Endowment*, 4(3), 2010.

[11] Jiyang Bai, Yuxiang Ren, and Jiawei Zhang. Ripple walk training: A subgraph-based training framework for large and deep graph neural network. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2021.

[12] Paolo Bajardi, Alain Barrat, Lara Savini, and Vittoria Colizza. Optimizing surveillance for livestock disease spreading through animal movements. *Journal of the Royal Society Interface*, 9 (76):2814–2825, 2012.

[13] Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz. Communication-optimal parallel algorithm for Strassen's matrix multiplication. In *Proceedings of the Twenty-Fourth annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 193–204, 2012.

[14] Neil Band. MemFlow: Memory-aware distributed deep learning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2883–2885, 2020.

[15] Vishwash Batra, Aparajita Haldar, Yulan He, Hakan Ferhatosmanoglu, George Vogiatzis, and Tanaya Guha. Variational recurrent sequence-to-sequence retrieval for stepwise illustration. In *European Conference on Information Retrieval*, pages 50–64. Springer, 2020.

[16] Austin R Benson, Rediet Abebe, Michael T Schaub, Ali Jadbabaie, and Jon Kleinberg. Simplicial closure and higher-order link prediction. *Proceedings of the National Academy of Sciences*, 115 (48):E11221–E11230, 2018.

[17] Seth G Benzell, Avinash Collis, and Christos Nicolaides. Rationing social contact during the COVID-19 pandemic: Transmission risk and social benefits of US locations. *Proceedings of the National Academy of Sciences*, 117(26):14642–14644, 2020.

[18] Claude Berge. *Hypergraphs: combinatorics of finite sets*, volume 45. Elsevier, 1984.

[19] Christian Borgs, Michael Brautbar, Jennifer Chayes, and Brendan Lucier. Maximizing social influence in nearly optimal time. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 946–957. SIAM, 2014.

[20] Jeremy T Bradley, Douglas V de Jager, William J Knottenbelt, and Aleksandar Trifunović. Hypergraph partitioning for faster parallel PageRank computation. In *Formal Techniques for Computer Systems and Business Processes*, pages 155–171. Springer, 2005.

[21] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*, 2013.

[22] Thang Nguyen Bui and Curt Jones. A heuristic for reducing fill-in in sparse matrix factorization. Technical report, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1993.

[23] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. *Algorithm Engineering*, pages 117–158, 2016.

[24] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. DGCL: An efficient communication library for distributed GNN training. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 130–144, 2021.

[25] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[26] Umit V Catalyurek and Cevdet Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999.

[27] Ümit V Çatalyürek and Cevdet Aykanat. PaToH: Partitioning tool for hypergraphs. In *Encyclopedia of Parallel Computing*, pages 1479–1487. Springer, 2011.

[28] Serina Chang, Emma Pierson, Pang Wei Koh, Jaline Gerardin, Beth Redbird, David Grusky, and Jure Leskovec. Mobility network models of COVID-19 explain inequities and inform reopening. *Nature*, 589(7840):82–87, 2021.

[29] Hu Chen, Wenguang Chen, Jian Huang, Bob Robert, and Harold Kuhn. MPIPP: An automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters. In *Proceedings of the 20th Annual International Conference on Supercomputing*, pages 353–360, 2006.

[30] Jie Chen, Tengfei Ma, and Cao Xiao. FastGCN: Fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247*, 2018.

[31] Wei Chen, Yajun Wang, and Siyu Yang. Efficient influence maximization in social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge Discovery and Data Mining*, pages 199–208, 2009.

[32] Wei Chen, Yifei Yuan, and Li Zhang. Scalable influence maximization in social networks under the linear threshold model. In *2010 IEEE international Conference on Data Mining*, pages 88–97. IEEE, 2010.

[33] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-GCN: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 257–266, 2019.

[34] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at Facebook-scale. *Proceedings of the VLDB Endowment*, 8 (12):1804–1815, 2015.

[35] Nicholas A Christakis and James H Fowler. Social network sensors for early detection of contagious outbreaks. *PLoS ONE*, 5(9):e12948, 2010.

[36] Timothy A Davis. Algorithm 1000: Suitesparse: GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Transactions on Mathematical Software (TOMS)*, 45(4):1–25, 2019.

[37] Guilherme Ferraz de Arruda, Giovanni Petri, and Yamir Moreno. Social contagion models on hypergraphs. *Physical Review Research*, 2(2):023032, 2020.

[38] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. *Advances in Neural Information Processing Systems*, 29, 2016.

[39] Gunduz Demirci, Aparajita Haldar, and Hakan Ferhatosmanoglu. Scalable graph convolutional network training on distributed-memory systems. *Proceedings of the VLDB Endowment*, 16, 2022.

[40] Gunduz Vehbi Demirci and Hakan Ferhatosmanoglu. Partitioning sparse deep neural networks for scalable training and inference. In *Proceedings of the ACM International Conference on Supercomputing*, pages 254–265, 2021.

[41] Gianluca Dini, Marco Pelagatti, and Ida Maria Savino. An algorithm for reconnecting wireless sensor network partitions. In *European Conference on Wireless Sensor Networks*, pages 253–267. Springer, 2008.

[42] Pedro Domingos and Matt Richardson. Mining the network value of customers. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 57–66, 2001.

[43] Ulrich Elsner. Graph partitioning - a survey. 1997.

[44] Wenfei Fan, Muyang Liu, Chao Tian, Ruiqi Xu, and Jingren Zhou. Incrementalization of graph partitioning algorithms. *Proceedings of the VLDB Endowment*, 13(8):1261–1274, 2020.

[45] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, et al. GraphScope: A unified engine for big graph processing. *Proceedings of the VLDB Endowment*, 14(12):2879–2892, 2021.

[46] Wenfei Fan, Ruiqi Xu, Qiang Yin, Wenyuan Yu, and Jingren Zhou. Application-driven graph partitioning. *The VLDB Journal*, pages 1–24, 2022.

[47] Jiawei Fei, Chen-Yu Ho, Atal N Sahu, Marco Canini, and Amedeo Sapio. Efficient sparse collective communication and its application to accelerate distributed deep learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 676–691, 2021.

[48] Song Feng, Emily Heath, Brett Jefferson, Cliff Joslyn, Henry Kvinge, Hugh D Mitchell, Brenda Praggastis, Amie J Eisfeld, Amy C Sims, Larissa B Thackray, et al. Hypergraph models of biological networks to identify genes critical to pathogenic viral response. *BMC Bioinformatics*, 22(1):1–21, 2021.

[49] Yifan Feng, Haoxuan You, Zizhao Zhang, Rongrong Ji, and Yue Gao. Hypergraph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3558–3565, 2019.

[50] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[51] CM Fiduccia and RM Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, pages 175–181, 1982.

[52] Thomas Gaudelet, Ben Day, Arian R Jamasb, Jyothish Soman, Cristian Regep, Gertrude Liu, Jeremy BR Hayter, Richard Vickers, Charles Roberts, Jian Tang, et al. Utilizing graph machine learning within drug discovery and development. *Briefings in Bioinformatics*, 22(6), 2021.

[53] Nathalie TH Gayraud, Evaggelia Pitoura, and Panayiotis Tsaparas. Diffusion maximization in evolving social networks. In *Proceedings of the 2015 ACM Conference on Online Social Networks*, pages 125–135, 2015.

[54] Mathieu Génois and Alain Barrat. Can co-location be used as a proxy for face-to-face contacts? *EPJ Data Science*, 7(1):1–18, 2018.

[55] Gurbinder Gill, Roshan Dathathri, Loc Hoang, and Keshav Pingali. A study of partitioning policies for graph analytics on large-scale distributed platforms. *Proceedings of the VLDB Endowment*, 12 (4):321–334, 2018.

[56] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International Conference on Machine Learning*, pages 1263–1272. PMLR, 2017.

[57] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, pages 17–30, 2012.

[58] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Scalable shared-memory hypergraph partitioning. In *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 16–30. SIAM, 2021.

[59] Andrzej Grabowski and Andrzej Jarynowski. Rumor propagation in temporal contact network from Polish polls. In *2016 Third European Network Intelligence Conference (ENIC)*, pages 85–89. IEEE, 2016.

[60] Bnaya Gross, Zhiguo Zheng, Shiyan Liu, Xiaoqi Chen, Alon Sela, Jianxin Li, Daqing Li, and Shlomo Havlin. Spatio-temporal propagation of COVID-19 pandemics. *EPL (Europhysics Letters)*, 131(5):58003, 2020.

[61] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 855–864, 2016.

[62] Adrien Guille, Hakim Hacid, Cecile Favre, and Djamel A Zighed. Information diffusion in online social networks: A survey. *ACM Sigmod Record*, 42(2):17–28, 2013.

[63] Rustam Guliyev, Aparajita Haldar, and Hakan Ferhatosmanoglu. D3-GNN: Dynamic distributed dataflow for streaming graph neural networks. 2022.

[64] Lars Hagen and Andrew B Kahng. New spectral methods for ratio cut partitioning and clustering. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(9):1074–1085, 1992.

[65] Aparajita Haldar, Teddy Cunningham, and Hakan Ferhatosmanoglu. RAGUEL: Recourse-aware group unfairness elimination. In *Proceedings of the 31st ACM International Conference on Information and Knowledge Management*, pages 666–675, 2022.

[66] Aparajita Haldar, Shuang Wang, Gunduz Vehbi Demirci, Joe Oakley, and Hakan Ferhatosmanoglu. Temporal cascade model for analyzing spread in evolving networks. *ACM Transactions on Spatial Algorithms and Systems*, 2023.

[67] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in Neural Information Processing Systems*, 30, 2017.

[68] Kai Han, Keke Huang, Xiaokui Xiao, Jing Tang, Aixin Sun, and Xueyan Tang. Efficient algorithms for adaptive influence maximization. *Proceedings of the VLDB Endowment*, 11(9):1029–1040, 2018.

[69] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M Tamer Özsu, Xingfang Wang, and Tianqi Jin. An experimental comparison of Pregel-like graph processing systems. *Proceedings of the VLDB Endowment*, 7(12):1047–1058, 2014.

[70] Bruce Hendrickson and Tamara G Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26(12):1519–1534, 2000.

[71] Bruce Hendrickson and Tamara G Kolda. Partitioning rectangular and structurally unsymmetric sparse matrices for parallel processing. *SIAM Journal on Scientific Computing*, 21(6):2048–2072, 2000.

[72] Herbert W Hethcote. The mathematics of infectious diseases. *SIAM Review*, 42(4):599–653, 2000.

[73] Petter Holme. Three faces of node importance in network epidemiology: Exact results for small graphs. *Physical Review E*, 96(6):062305, 2017.

[74] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. FeatGraph: A flexible and efficient backend for graph neural network systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2020.

[75] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. GE-SpMM: General-purpose sparse matrix-matrix multiplication on GPUs for graph neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2020.

[76] Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. Adaptive sampling towards fast graph representation learning. *Advances in Neural Information Processing Systems*, 31, 2018.

[77] Yuchi Huang, Qingshan Liu, Shaoting Zhang, and Dimitris N Metaxas. Image retrieval via probabilistic hypergraph ranking. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 3376–3383. IEEE, 2010.

[78] Edmund Ihler, Dorothea Wagner, and Frank Wagner. Modeling hypergraphs by graphs with the same mincut properties. *Information Processing Letters*, 45(4):171–175, 1993.

[79] A Jain, I Liu, A Sarda, and P Molino. Food discovery with Uber Eats: Using graph learning to power recommendations, 2019.

[80] Glen Jeh and Jennifer Widom. SimRank: A measure of structural-context similarity. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 538–543, 2002.

[81] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the accuracy, scalability, and performance of graph neural networks with Roc. *Proceedings of Machine Learning and Systems*, 2:187–198, 2020.

[82] Weiwei Jiang and Jiayun Luo. Graph neural network for traffic forecasting: A survey. *Expert Systems with Applications*, page 117921, 2022.

[83] Ruoming Jin, Victor E Lee, and Longjie Li. Scalable and axiomatic ranking of network role similarity. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 8(1):1–37, 2014.

[84] George Karypis and Vipin Kumar. METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. 1997.

[85] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[86] David Kempe, Jon Kleinberg, and Éva Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 137–146, 2003.

[87] Brian W Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, 1970.

[88] Jinha Kim, Wonyeol Lee, and Hwanjo Yu. CT-IC: Continuously activated and time-restricted independent cascade model for viral marketing. *Knowledge-Based Systems*, 62:57–68, 2014.

[89] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[90] István Z Kiss, Joel C Miller, Péter L Simon, et al. Mathematics of epidemics on networks. *Cham: Springer*, 598, 2017.

[91] Stephen M Kissler, Petra Klepac, Maria Tang, Andrew JK Conlan, and Julia R Gog. Sparking "the BBC Four Pandemic": Leveraging citizen science and mobile phones to model the spread of disease. *bioRxiv*, page 479154, 2020.

[92] Johannes Klicpera, Aleksandar Bojchevski, and Stephan Günnemann. Predict then propagate: Graph neural networks meet personalized PageRank. *arXiv preprint arXiv:1810.05997*, 2018.

[93] Penporn Koanantakool, Ariful Azad, Aydin Buluç, Dmitriy Morozov, Sang-Yun Oh, Leonid Oliker, and Katherine Yelick. Communication-avoiding parallel sparse-dense matrix-matrix multiplication. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 842–853. IEEE, 2016.

[94] Elena V Konstantinova and Vladimir A Skorobogatov. Application of hypergraph theory in chemistry. *Discrete Mathematics*, 235(1-3):365–383, 2001.

[95] Kelly Kostopoulou, Hang Xu, Aritra Dutta, Xin Li, Alexandros Ntoulas, and Panos Kalnis. DeepReduce: A sparse-tensor communication framework for distributed deep learning. *arXiv preprint arXiv:2102.03112*, 2021.

[96] A Kumar, S Nakandala, Y Zhang, S Li, A Gemawat, and K Nagrecha. Cerebro: A layered data platform for scalable deep learning. In *11th Annual Conference on Innovative Data Systems Research (CIDR '21)*, 2021.

[97] Srijan Kumar, William L Hamilton, Jure Leskovec, and Dan Jurafsky. Community interaction and conflict on the web. In *Proceedings of the 2018 World Wide Web Conference*, pages 933–943, 2018.

[98] Rakshith Kunchum, Ankur Chaudhry, Aravind Sukumaran-Rajam, Qingpeng Niu, Israt Nisa, and P Sadayappan. On improving performance of sparse matrix-matrix multiplication on GPUs. In *Proceedings of the International Conference on Supercomputing*, pages 1–11, 2017.

[99] Mitsuru Kusumoto, Takanori Maehara, and Ken-ichi Kawarabayashi. Scalable similarity search for SimRank. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 325–336, 2014.

[100] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, pages 177–187, 2005.

[101] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne VanBriesen, and Natalie Glance. Cost-effective outbreak detection in networks. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 420–429, 2007.

[102] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Predicting positive and negative links in online social networks. In *Proceedings of the 19th International Conference on World Wide Web*, pages 641–650, 2010.

[103] Pei Li, Hongyan Liu, Jeffrey Xu Yu, Jun He, and Xiaoyong Du. Fast single-pair simrank computation. In *Proceedings of the 2010 SIAM International Conference on Data Mining*, pages 571–582. SIAM, 2010.

[104] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

[105] Zhenjiang Lin, Michael R Lyu, and Irwin King. MatchSim: A novel similarity measure based on maximum neighborhood matching. *Knowledge and Information Systems*, 32(1):141–166, 2012.

[106] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. PaGraph: Scaling GNN training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 401–415, 2020.

[107] Bo Liu, Gao Cong, Dong Xu, and Yifeng Zeng. Time constrained influence maximization in social networks. In *Proceedings of the 2012 IEEE 12th International Conference on Data Mining*, pages 439–448, 2012.

[108] Husong Liu, Shengliang Lu, Xinyu Chen, and Bingsheng He. G3: When graph neural networks meet parallel graph processing systems on GPUs. *Proceedings of the VLDB Endowment*, 13(12): 2813–2816, 2020.

[109] Xiaodong Liu, Mo Li, Shanshan Li, Shaoliang Peng, Xiangke Liao, and Xiaopei Lu. IMGPU: GPU-accelerated influence maximization in large-scale social networks. *IEEE Transactions on Parallel and distributed Systems*, 25(1):136–145, 2013.

[110] Dmitry Lizorkin, Pavel Velikhov, Maxim Grinev, and Denis Turdakov. Accuracy estimate and optimization techniques for SimRank computation. *The VLDB Journal*, 19(1):45–66, 2010.

[111] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. Large-scale distributed graph computing systems: An experimental evaluation. *Proceedings of the VLDB Endowment*, 8(3):281–292, 2014.

[112] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. Neugraph: parallel deep neural network computation on large graphs. In *Proceedings of the 2019 USENIX Annual Technical Conference*, pages 443–457, 2019.

[113] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 135–146, 2010.

[114] Christian Mayer, Ruben Mayer, Sukanya Bhowmik, Lukas Epple, and Kurt Rothermel. HYPE: Massive hypergraph partitioning with neighborhood expansion. In *2018 IEEE International Conference on Big Data*, pages 458–467. IEEE, 2018.

[115] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj Kalamkar, Nesreen K Ahmed, and Sasikanth Avancha. DistGNN: Scalable distributed training for large-scale graph neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[116] Xupeng Miao, Hailin Zhang, Yining Shi, Xiaonan Nie, Zhi Yang, Yangyu Tao, and Bin Cui. HET: scaling out huge embedding model training via cache-enabled distributed framework. *Proceedings of the VLDB Endowment*, 15(2):312–320, 2021.

[117] Xupeng Miao, Wentao Zhang, Yingxia Shao, Bin Cui, Lei Chen, Ce Zhang, and Jiawei Jiang. Lasagne: A multi-layer graph convolutional network framework via node-aware deep architecture. *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[118] Alessio Micheli. Neural network for graphs: A contextual constructive approach. *IEEE Transactions on Neural Networks*, 20(3):498–511, 2009.

[119] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. Large graph convolutional network training with GPU-oriented data communication architecture. *Proceedings of the VLDB Endowment*, 14(11):2087–2100, 2021.

[120] Rajat Mittal, Charles Meneveau, and Wen Wu. A mathematical framework for estimating risk of airborne transmission of COVID-19 with application to face mask use and social distancing. *Physics of Fluids*, 32(10):101903, 2020.

[121] Federico Monti, Fabrizio Frasca, Davide Eynard, Damon Mannion, and Michael M Bronstein. Fake news detection on social media using geometric deep learning. *arXiv preprint arXiv:1902.06673*, 2019.

[122] Naoto Ohsaka, Takuya Akiba, Yuichi Yoshida, and Ken-ichi Kawarabayashi. Dynamic influence analysis in evolving networks. *Proceedings of the VLDB Endowment*, 9(12):1077–1088, 2016.

[123] Chao-Wei Ou and Sanjay Ranka. Parallel incremental graph partitioning. *IEEE Transactions on Parallel and Distributed Systems*, 8(8):884–896, 1997.

[124] Anil Pacaci and M Tamer Özsu. Experimental analysis of streaming algorithms for graph partitioning. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1375–1392, 2019.

[125] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[126] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao Schardl, and Charles Leiserson. EvolveGCN: Evolving graph convolutional networks for dynamic graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 5363–5370, 2020.

[127] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. Sancus: staleness-aware communication-avoiding full-graph decentralized training in large-scale graph neural networks. *Proceedings of the VLDB Endowment*, 15(9):1937–1950, 2022.

[128] Emanuele Pepe, Paolo Bajardi, Laetitia Gauvin, Filippo Privitera, Brennan Lake, Ciro Cattuto, and Michele Tizzoni. COVID-19 outbreak response, a dataset to assess mobility changes in Italy following national lockdown. *Scientific Data*, 7(1):1–7, 2020.

[129] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. DeepWalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 701–710, 2014.

[130] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, and Giorgio Iacoboni. HDRF: Stream-based partitioning for power-law graphs. In *Proceedings of the 24th ACM International Conference on Information and Knowledge Management*, pages 243–252, 2015.

[131] B Aditya Prakash, Hanghang Tong, Nicholas Valler, Michalis Faloutsos, and Christos Faloutsos. Virus propagation on time-varying networks: Theory and immunization algorithms. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 99–114. Springer, 2010.

[132] Md Khaledur Rahman, Majedul Haque Sujon, and Ariful Azad. FusedMM: A unified SDDMM-SpMM kernel for graph embedding and graph neural networks. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 256–266. IEEE, 2021.

[133] Wendi Ren, Jiajing Wu, Xi Zhang, Rong Lai, and Liang Chen. A stochastic model of cascading failure dynamics in communication networks. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 65(5):632–636, 2018.

[134] Sungmin Rhee, Seokjun Seo, and Sun Kim. Hybrid approach of relation network and localized graph convolutional filtering for breast cancer subtype classification. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 3527–3534, 2018.

[135] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. Temporal graph networks for deep learning on dynamic graphs. *arXiv preprint arXiv:2006.10637*, 2020.

[136] Sascha Rothe and Hinrich Schütze. CoSimRank: A flexible & efficient graph-theoretic similarity measure. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1392–1402, 2014.

[137] Alvaro Sanchez-Gonzalez, Nicolas Heess, Jost Tobias Springenberg, Josh Merel, Martin Riedmiller, Raia Hadsell, and Peter Battaglia. Graph networks as learnable physics engines for inference and control. In *International Conference on Machine Learning*, pages 4470–4479. PMLR, 2018.

[138] Mohamed Sarwat, Sameh Elnikety, Yuxiong He, and Mohamed F Mokbel. Horton+: A distributed system for processing declarative reachability queries over partitioned graphs. *Proceedings of the VLDB Endowment*, 6(14):1918–1929, 2013.

[139] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. High-quality hypergraph partitioning. *ACM Journal of Experimental Algorithms (JEA)*, 2020.

[140] Oguz Selvitopi, Benjamin Brock, Israt Nisa, Alok Tripathy, Katherine Yelick, and Aydın Buluç. Distributed-memory parallel algorithms for sparse times tall-skinny-dense matrix multiplication. In *Proceedings of the ACM International Conference on Supercomputing*, pages 431–442, 2021.

[141] R Oguz Selvitopi, Ata Turk, and Cevdet Aykanat. Replicated partitioning for undirected hypergraphs. *Journal of Parallel and Distributed Computing*, 72(4):547–563, 2012.

[142] Marco Serafini. Scalable graph neural network training: The case for sampling. *ACM SIGOPS Operating Systems Review*, 55(1):68–76, 2021.

[143] Kai Shu, H Russell Bernard, and Huan Liu. Studying fake news via network analysis: Detection and mitigation. In *Emerging Research Challenges and Opportunities in Computational Social Network Analysis and Mining*, pages 43–65. Springer, 2019.

[144] David I Shuman, Sunil K Narang, Pascal Frossard, Antonio Ortega, and Pierre Vandergheynst. The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains. *IEEE Signal Processing Magazine*, 30(3):83–98, 2013.

[145] Guojie Song, Xiabing Zhou, Yu Wang, and Kunqing Xie. Influence maximization on large-scale mobile social network: a divide-and-conquer method. *IEEE Transactions on Parallel and Distributed Systems*, 26(5):1379–1392, 2014.

[146] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1222–1230, 2012.

[147] Zoya Svitkina and Éva Tardos. Min-max multiway cut. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 207–218. Springer, 2004.

[148] Anshoo Tandon, Teng Joon Lim, and Utku Tefek. Sentinel based malicious relay detection in wireless IoT networks. *Journal of Communications and Networks*, 21(5):458–468, 2019.

[149] Jing Tang, Xueyan Tang, Xiaokui Xiao, and Junsong Yuan. Online processing algorithms for influence maximization. In *Proceedings of the 2018 International Conference on Management of Data*, pages 991–1005, 2018.

[150] Youze Tang, Xiaokui Xiao, and Yanchen Shi. Influence maximization: Near-optimal time complexity meets practical efficiency. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 75–86, 2014.

[151] Fatih Taşyaran, Berkay Demireller, Kamer Kaya, and Bora Uçar. Streaming hypergraph partitioning algorithms on limited memory environments. In *International Conference on High Performance Computing and Simulation (HPCS 2020)*, pages 1–8. IEEE, 2021.

[152] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, et al. Dorylus: Affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2021)*, pages 495–514, 2021.

[153] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From “think like a vertex” to “think like a graph”. *Proceedings of the VLDB Endowment*, 7(3): 193–204, 2013.

[154] Alok Tripathy, Katherine Yelick, and Aydin Buluc. Reducing communication in graph neural network training. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2020)*, pages 987–1000. IEEE Computer Society.

[155] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. FENNEL: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, pages 333–342, 2014.

[156] Luis M Vaquero, Felix Cuadrado, Dionysios Logothetis, and Claudio Martella. Adaptive partitioning for large-scale dynamic graphs. In *Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems*, pages 144–153, 2014.

[157] Vijay V Vazirani. *Approximation algorithms*, volume 1. Springer, 2001.

[158] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[159] Shiv Verma, Luke M Leslie, Yosub Shin, and Indranil Gupta. An experimental comparison of partitioning strategies in distributed graph processing. *Proceedings of the VLDB Endowment*, 10 (5):493–504, 2017.

[160] Hanzhi Wang, Zhewei Wei, Junhao Gan, Sibo Wang, and Zengfeng Huang. Personalized PageRank to a target node, revisited. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 657–667, 2020.

[161] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. Billion-scale commodity embedding for e-commerce recommendation in Alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 839–848, 2018.

[162] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, et al. Deep Graph Library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.

[163] Wei Wang, Meihui Zhang, Gang Chen, HV Jagadish, Beng Chin Ooi, and Kian-Lee Tan. Database meets deep learning: Challenges and opportunities. *ACM SIGMOD Record*, 45(2):17–22, 2016.

[164] Xiaoyang Wang, Yao Ma, Yiqi Wang, Wei Jin, Xin Wang, Jiliang Tang, Caiyan Jia, and Jian Yu. Traffic flow prediction via spatial temporal graph neural network. In *Proceedings of the Web Conference 2020*, pages 1082–1092, 2020.

[165] Yingzi Wang, Xiao Zhou, Anastasios Noulas, Cecilia Mascolo, Xing Xie, and Enhong Chen. Predicting the spatio-temporal evolution of chronic diseases in population with human mobility data. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 3578–3584, 2018.

[166] Zhouxia Wang, Tianshui Chen, Jimmy Ren, Weihao Yu, Hui Cheng, and Liang Lin. Deep reasoning with knowledge graph for social relationship understanding. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1021–1028, 2018.

[167] Brendan Whitaker, Denis Newman-Griffis, Aparajita Haldar, Hakan Ferhatosmanoglu, and Eric Fosler-Lussier. Characterizing the impact of geometric properties of word embeddings on task performance. *NAACL HLT 2019*, page 8, 2019.

[168] Michael M Wolf, Alicia M Klinvex, and Daniel M Dunlavy. Advantages to modeling relational data using hypergraphs versus graphs. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2016.

[169] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. Simplifying graph convolutional networks. In *International Conference on Machine Learning*, pages 6861–6871. PMLR, 2019.

[170] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, 2020.

[171] Xin Xia, Hongzhi Yin, Junliang Yu, Qinyong Wang, Lizhen Cui, and Xiangliang Zhang. Self-supervised hypergraph convolutional networks for session-based recommendation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 4503–4511, 2021.

[172] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. GraphX: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, pages 1–6, 2013.

[173] Naganand Yadati, Madhav Nimishakavi, Prateek Yadav, Vikram Nitin, Anand Louis, and Partha Talukdar. HyperGCN: A new method for training graph convolutional networks on hypergraphs. *Advances in Neural Information Processing Systems*, 32, 2019.

[174] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment*, 7(14):1981–1992, 2014.

[175] Dingqi Yang, Daqing Zhang, Vincent W Zheng, and Zhiyong Yu. Modeling user activity preference by leveraging user spatial temporal characteristics in LBSNs. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 45(1):129–142, 2014.

[176] Hongxia Yang. Aligraph: A comprehensive graph neural network platform. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 3165–3166, 2019.

[177] Shengqi Yang, Xifeng Yan, Bo Zong, and Arijit Khan. Towards effective partition management for large graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 517–528, 2012.

[178] Yu Yang, Zhefeng Wang, Jian Pei, and Enhong Chen. Tracking influential individuals in dynamic networks. *IEEE Transactions on Knowledge and Data Engineering*, 29(11):2615–2628, 2017.

[179] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 974–983, 2018.

[180] Bing Yu, Haoteng Yin, and Zhanxing Zhu. Spatio-temporal graph convolutional networks: a deep learning framework for traffic forecasting. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 3634–3640, 2018.

[181] Junliang Yu, Hongzhi Yin, Jundong Li, Qinyong Wang, Nguyen Quoc Viet Hung, and Xiangliang Zhang. Self-supervised multi-channel hypergraph convolutional network for social recommendation. In *Proceedings of the Web Conference 2021*, pages 413–424, 2021.

[182] Weiren Yu and Julie A McCann. Efficient partial-pairs SimRank search on large networks. *Proceedings of the VLDB Endowment*, 8(5):569–580, 2015.

[183] Weiren Yu, Sima Iranmanesh, Aparajita Haldar, Maoyin Zhang, and Hakan Ferhatosmanoglu. An axiomatic role similarity measure based on graph topology. In *Software Foundations for Data Interoperability and Large Scale Graph Data Analytics*, pages 33–48. Springer, 2020.

[184] Weiren Yu, Sima Iranmanesh, Aparajita Haldar, Maoyin Zhang, and Hakan Ferhatosmanoglu. RoleSim*: Scaling axiomatic role-based similarity ranking on large graphs. *World Wide Web*, 25 (2):785–829, 2022.

[185] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache Spark: A unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[186] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi. AGL: A scalable system for industrial-purpose graph machine learning. *Proceedings of the VLDB Endowment*, 13(12):3125–3137, 2020.

[187] Si Zhang, Hanghang Tong, Jiejun Xu, and Ross Maciejewski. Graph convolutional networks: a comprehensive review. *Computational Social Networks*, 6(1):1–23, 2019.

[188] Yuhao Zhang, Frank Mcquillan, Nandish Jayaram, Nikhil Kak, Ekta Khanna, Orhan Kislal, Domino Valdano, and Arun Kumar. Distributed deep learning on data systems: A comparative analysis of approaches. *Proceedings of the VLDB Endowment*, 14(10):1769–1782, 2021.

[189] Zhipeng Zhang, Yingxia Shao, Bin Cui, and Ce Zhang. An experimental evaluation of SimRank-based similarity search algorithms. *Proceedings of the VLDB Endowment*, 10(5):601–612, 2017.

[190] Ziwei Zhang, Peng Cui, and Wenwu Zhu. Deep learning on graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[191] Peixiang Zhao, Jiawei Han, and Yizhou Sun. P-Rank: A comprehensive structural similarity measure over information networks. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, pages 553–562, 2009.

[192] Xiang Zhao, Chuan Xiao, Xuemin Lin, Qing Liu, and Wenjie Zhang. A partition-based approach to structure similarity search. *Proceedings of the VLDB Endowment*, 7(3):169–180, 2013.

[193] Chenguang Zheng, Hongzhi Chen, Yuxuan Cheng, Zhezheng Song, Yifan Wu, Changji Li, James Cheng, Hao Yang, and Shuai Zhang. ByteGNN: Efficient graph neural network training at large scale. *Proceedings of the VLDB Endowment*, 15(6):1228–1242, 2022.

[194] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. DistDGL: Distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 36–44. IEEE, 2020.

[195] Da Zheng, Xiang Song, Chengru Yang, Dominique LaSalle, and George Karypis. Distributed hybrid CPU and GPU training for graph neural networks on billion-scale heterogeneous graphs. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 4582–4591, 2022.

[196] Dengyong Zhou, Jiayuan Huang, and Bernhard Schölkopf. Learning with hypergraphs: Clustering, classification, and embedding. *Advances in Neural Information Processing Systems*, 19, 2006.

[197] Hongkuan Zhou, Ajitesh Srivastava, Hanqing Zeng, Rajgopal Kannan, and Viktor Prasanna. Accelerating large scale real-time GNN inference using channel pruning. *Proceedings of the VLDB Endowment*, 14(9):1597–1605, 2021.

[198] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

[199] Qingyi Zhu, Xiaofan Yang, and Jianguo Ren. Modeling and analysis of the spread of computer virus. *Communications in Nonlinear Science and Numerical Simulation*, 17(12):5117–5124, 2012.

[200] Jason Y Zien, Martine DF Schlag, and Pak K Chan. Multilevel spectral hypergraph partitioning with arbitrary vertex sizes. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(9):1389–1399, 1999.