Gossip-Based Algorithms for Information Dissemination and Graph Clustering

FATEMEH RAHIMIAN

Doctoral Thesis in
Information and Communication Technology
Stockholm, Sweden 2014
Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie doktorandexamen i datalogi torsdag den 22 maj 2014 klockan 13:00 i sal e i forum IT-Universitetet, Kungl Tekniskahögskolan, Isafjordsgatan 39, Kista.

Swedish Institute of Computer Science
SICS Dissertation Series 68
ISSN 1101-1335.

© Fatemeh Rahimian, May, 2014

Tryck: Universitetsservice US AB
Decentralized algorithms are becoming ever more prevalent in almost all real-world applications that are either data intensive, computation intensive or both. This thesis presents a few decentralized solutions for large-scale (i) data dissemination, (ii) graph partitioning, and (iii) data disambiguation. All these solutions are based on gossip, a light weight peer-to-peer data exchange protocol, and thus, appropriate for execution in a distributed environment.

For efficient data dissemination, we make use of the publish/subscribe communication model and provide two distributed solutions, one for topic-based and one for content-based subscriptions, named Vitis and Vinifera respectively. These systems propagate large quantities of data to interested users with a relatively low overhead. Without any central coordinator and only with the use of gossip, we build a novel topology that enables efficient routing in an unstructured overlay. We construct a hybrid system by injecting structure into an otherwise unstructured network. The resulting structure resembles a navigable small-world network that spans along clusters of nodes that have similar subscriptions. The properties of such an overlay make it an ideal platform for efficient data dissemination in large-scale systems. Our solutions significantly outperforms their counterparts on various subscription and churn scenarios, from both synthetic models and real-world traces.

We then investigate how gossiping protocols can be used, not for overlay construction, but for operating on fixed overlay topologies, which resemble graphs. In particular we study the NP-Complete problem of graph partitioning and present a distributed partitioning solution for very large graphs. This solution, called Ja-be-Ja, is based on local search and does not require access to the entire graph simultaneously. It is, therefore, appropriate for graphs that can not even fit into the memory of a single computer. Once again gossip-based algorithms prove efficient as they enable implementing light-weight peer sampling services, which supply graph nodes with partial knowledge about other nodes in the graph. The performance of our partitioning algorithm is comparable to centralized graph partitioning algorithms, and yet it is scalable and can be executed on several machines in parallel or even in a completely distributed peer-to-peer overlay. It can be used for both edge-cut and vertex-cut partitioning of graphs and can produce partition sizes of any given distribution.

We further extend the use of gossiping protocols to find natural clusters in a graph instead of producing a given number of partitions. This problem, known as graph community detection, has extensive application in various fields and communities. We take the use of our community detection algorithm to the realm of linguistics and address a well-known problem of data disambiguation. In particular, we provide a parallel community detection algorithm for cross-document coreference problem. We operate on graphs that we construct by representing documents’ keywords as nodes and the co-location of those keywords in a document as edges. We then exploit the particular nature of such graphs, which is coreferent words are topologically clustered, and thus, can be efficiently discovered by our community detection algorithm.
To Amir Hossein
Acknowledgements

It is a long journey, the life of a PhD student, full of ups and downs, joys and disappointments, failures and successes. But you should never lose hope, specially if you are, like me, surrounded with amazing people who generously give their time, support and advice to you.

Foremost, I would like to express my deepest gratitude to my advisor, Professor Seif Haridi. Your broad knowledge and expertise, your incredible enthusiasm and your continuous fatherly support, provided me with a great opportunity and excellent atmosphere for doing research. Without you this dissertation would never have been materialized. I will be forever indebted to you. Thank you sir!

I would like to thank my second advisor Dr. Šarūnas Girdzijauskas. I could not imagine having a better mentor than you. You generously gave me all the time and support I needed, showed me how to do research and gave me the most insightful guidance. Words can not explain how grateful I am to you, for you bore with me through all ups and downs. Labai ačiū!

I also would like to thank my co-advisor, Dr. Jim Dowling. I am extremely grateful to you for the illuminating discussions, your constructive feedbacks and invaluable support. Go raibh maith agat!

My deepest appreciation goes to Dr. Amir Hossein Payberah. I am thankful to you not only for your enormous contributions to the work presented in this thesis, but also for your generous help and support during the whole course of my studies. Yek donya Mamnoonam! I dedicate this work to you with all my love and gratitude.

I would like to thank Professor Márk Jelasity for our collaboration on graph partitioning. It has been an honor working with you and benefiting from your extensive knowledge. Köszönöm!

I am very much obliged to Professor Amr El-Abbadi and Dr. Ceren Budak for broadening my knowledge in the field of streaming data processing. Thank you!

My special thanks go to Dr. Sverker Janson at SICS. You are an amazing manager, always full of inspiration and great advices. Tack så mycket!

I am grateful to Martin Neumann for many fruitful and enthusiastic discussions we had on the topic of graph processing. Dankeschön!

I am thankful for the assistance given to me by Thinh Le Neuyen Huu for the work on locality awareness in publish/subscribe systems.

I thank Dr. Vladimir Vlassov, Professor Christian Schulte, Professor Alberto Montresor, Dr. Ali Ghodsi, Dr. Tallat Shafaat, Dr. Cosmin Arad, Dr. Ahmad Al-Shishtawy, Dr. Ian Marsh, Dr. Nima Dokoohaki, Dr. Roberto Roverso, Dr. Raul Jimenez, Niklas Ekström and Alex Averbuch. It has been a great opportunity working along side you and benefiting from your advices, experiences and warm encouragements. Thank you all!

I am also obliged to my colleagues and fellow students at KTH and SICS, for contributing to such a great environment for doing research.

I gratefully acknowledge the financial support I have received from End-to-End Clouds project funded by the Swedish Foundation for Strategic Research (SSF)
under the contract RIT10-0043.

Big thanks to my dear friends, without whom I could not get through the hardships of this rather bumpy way. And finally, my dear family! How can I ever thank you? Your endless love and support fills me with life. You are my source of inspiration and I am forever indebted to you. Loads of love and thanks to you all!
Contents

I Thesis Overview 1

1 Introduction 3
  1.1 Models and Assumptions . . . . . . . . . . . . . . . . . . . . . . . . 5
  1.2 Outline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2 Background 7
  2.1 Peer-to-Peer Overlays . . . . . . . . . . . . . . . . . . . . . . . . . 7
  2.2 Gossiping Protocol . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
  2.3 Peer Sampling Services . . . . . . . . . . . . . . . . . . . . . . . . . 9
  2.4 Topology Management . . . . . . . . . . . . . . . . . . . . . . . . . 10
  2.5 Small-world Networks . . . . . . . . . . . . . . . . . . . . . . . . . . 11

3 Thesis contribution 13
  3.1 Information Dissemination . . . . . . . . . . . . . . . . . . . . . . . . 13
  3.2 Graph Partitioning . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
  3.3 Coreference Resolution . . . . . . . . . . . . . . . . . . . . . . . . . . 17
  3.4 Publications . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

4 Conclusions 21

II Research Papers 23

5 Vitis: A Gossip-based Hybrid Overlay for Internet-scale Publish/Subscribe 25
  5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
  5.2 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
  5.3 Vitis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
  5.4 Experiments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
CONTENTS

5.5 Conclusion ........................................... 47

6 Locality-awareness in a Peer-to-Peer Publish/Subscribe Network 51
   6.1 Introduction ....................................... 53
   6.2 Related Work ...................................... 54
   6.3 Locality-aware Publish/Subscribe ................... 56
   6.4 Evaluations ....................................... 60
   6.5 Conclusions ...................................... 65

7 Subscription Awareness Meets Rendezvous Routing 67
   7.1 Introduction ....................................... 69
   7.2 Related work ...................................... 72
   7.3 Architectural Model ................................. 74
   7.4 Vinifera ........................................... 74
   7.5 Evaluation ......................................... 82
   7.6 Conclusions ...................................... 87

8 JabeJa: A Distributed Algorithm for Balanced Graph Partitioning 89
   8.1 Introduction ....................................... 91
   8.2 Problem statement .................................. 93
   8.3 Related Work ...................................... 95
   8.4 Solution ........................................... 96
   8.5 Experimental evaluation ............................ 101
   8.6 Conclusions ...................................... 109

9 Distributed Vertex-Cut Partitioning 111
   9.1 Introduction ....................................... 113
   9.2 Problem statement .................................. 116
   9.3 Solution ........................................... 117
   9.4 Experiments ....................................... 121
   9.5 Related Work ...................................... 124
   9.6 Conclusions ...................................... 126

10 Parallel Community Detection For Cross-Document Coreference 127
   10.1 Introduction ...................................... 129
   10.2 Terminology ...................................... 132
   10.3 Solution ........................................... 133
   10.4 Experiments ...................................... 136
   10.5 Related Work ...................................... 140
   10.6 Conclusion and Future Work ....................... 143

Bibliography 145
Part I

Thesis Overview
Chapter 1

Introduction

“640K is enough for anyone, and by the way, what’s a network?”
- William Gates III, President of Microsoft Corporation, 1984

Far from the early predictions, in the world we live today, in the heart of almost every real-world application lies an essential need for big data management and decentralized computations over a network of computers.

Since the emergence of Web 2.0 applications, users of Internet do not merely consume data, but also produce data at a considerable rate. People share photos and videos on social networks, write blogs, even advertise and sell personal items on-line. The huge quantities of data that we have to deal with, does not contain independent and unrelated data items. Quite the contrary, data in today’s world is highly connected. The friends that we have in a social network connect us to the rest of the people. Our common interests group us together. Even far from the realm of social network, in a rather microscopic level of science, down to genomes and protein networks, we can observe connected data.

Networks are mathematically modelled with graphs. A graph consists of nodes/vertices and links/edges. For example, in a social network nodes represent users and links represent the friendship relations. In some other use cases graphs are to be constructed on the fly. For example, in a content dissemination network the links are established for data exchange. Whether a graph models the actual data or the path through which data can be transferred, it can grow very big. It may not even fit into the memory of a single computer. Graph databases, for instance, will soon require to span multiple machines not only for efficient storage and retrieval, but also for efficient computations over data. Needless to say, traditional centralized systems can no longer keep up to provide such resource demanding services. Also, centralized algorithms that require frequent global operations over the entire graph become prohibitively costly. Hence, decentralized solutions are becoming prevalent in most of real-world applications. More precisely, we need mechanism that replace the global operations with local ones and can work with only partial information.
 Thanks to the research in the field of distributed systems and peer-to-peer networks, such a mechanism exists. It is called gossiping, a.k.a. epidemic protocol. Gossiping is a light weight protocol for data exchange between neighboring nodes in a graph or network. Just like a virus can be transferred from one guy to another when they meet and spread in the whole society, two neighboring nodes of a graph can infect each other with pieces of information. Over time more and more nodes get infected and the information spreads in the graph. If these pieces of information are effectively aggregated at the nodes, they can be utilized to replace the global operations that exist in the centralized algorithms. In this thesis we employ gossiping to (i) build graphs, (ii) partition graphs, and (iii) extract information from the graphs.

First we study how we can employ gossiping, to construct overlays that provide large-scale publish/subscribe services. Publish/subscribe communication model is a ubiquitous protocol for data dissemination in today’s applications, in which users express their interest in form of subscriptions and get notified when some matching information is published. News syndication, multi-player games, social networks, and media streaming applications are a few examples of systems that are utilizing the publish/subscribe communication model. Depending on the application, the publish/subscribe service could be bandwidth intensive, as in streaming applications, or time critical, as in stock market applications, or may include a large number of subscriptions, as in social networks. We present two gossip-based distributed algorithms, namely Vitis and Vinifera for topic-based and content-based subscriptions, respectively. We use gossiping to cluster users with similar subscriptions together and at the same time embed a structure into this unstructured overlay, in order to make it navigable. The gossip protocol enables Vitis and Vinifera to operate without a central coordinator and any global knowledge.

Next, we use gossiping for partitioning a graph across multiple servers or clusters. This is a very important problem, because with the ever increasing size of the graphs, it is crucial to partition them into multiple smaller clusters that can be processed efficiently in parallel. Unlike the conventional parallel data processing, parallel graph processing requires each vertex or edge to be processed in the context of its neighborhood. Therefore, it is important to maintain the locality of information while partitioning the graph across multiple (virtual) machines. It is also important to produce equal size partitions that distribute the computational load evenly between clusters. We present a gossip-based distributed algorithm, called Ja-be-Ja for two types of graph partitioning, i.e., edge-cut and vertex-cut partitioning.

Finally, we use gossip for extracting information out of raw data. In particular, we address a linguistic problem that has to do with the classification of multiple documents with respect to an ambiguous term, such that each class of documents refer to a unique manifestation of the ambiguous term in reality. Note, this task may not always be a difficult task for humans. When one comes across Mercury in an article about the solar system, they instantly think of Mercury, the planet, and not about Mercury, the chemical element or Freddie Mercury. For a computer
though, such a disambiguation requires a considerable amount of processing. This problem, i.e., the task of disambiguating manifestations of real world entities in various records or mentions, is known as Entity Resolution or Coreference Resolution. We present a distributed solution to this problem, which is again based on gossip. We construct a graph out of the context words of the ambiguous mentions and then run a community detection algorithm to cluster this graph into several components, each referring to a distinct meaning of the ambiguous word.

Although all our solutions are based on gossip, they are not designed for exactly the same data distribution model. This is mainly due to the inherent differences in the problems in the first place. In the next section we introduce two models and point out which of our algorithms are designed for which model.

1.1 Models and Assumptions

In papers A, B and C, we frequently use the term node to refer to a processing unit, for example a computer in a peer-to-peer network. In papers D, E and F, node refers to graph nodes (or vertices). Since the peer-to-peer networks also resembles a graph, in the former case nodes can also be considered as nodes of a graph.

All the algorithms that are presented in this thesis work can be executed in a distributed environment. Using the graph processing terminology, all our algorithms are vertex-centric, meaning that they are executed by the graph nodes independently and in parallel. Two main models are considered:

**One-host-multiple-nodes**

This model is interesting for data centers or cloud environments, where each computer hosts thousands of nodes at the same time. The host periodically executes the algorithm over all the nodes that it holds. If an information exchange takes place with other nodes on the same host, the communication cost is negligible. However, information exchange across hosts is costly and constitutes the main body of the communication overhead.

**One-host-one-node**

In this model, each node could be placed either on a different host, or processed independently in a distributed framework. This model is appropriate for frameworks like GraphLab [1] or Pregel [2]. It can also be used in peer-to-peer (P2P) overlays, where each node is an independent computer. In both cases, no shared memory is required. Nodes communicate only using messages passing, and each message adds to the communication overhead.

Our work on publish/subscribe systems are designed for a peer-to-peer environment. Our partitioning algorithm can be executed in a one-host-one-node model, as well as the one-host-multiple-nodes model. Finally, our work on coreference resolution is designed for execution on one-host-multiple-nodes model. Note, the nature
of this problem does not match the one-host-one-node model. However, due to its complexity it requires parallel execution in face of large quantities of data. That is why our algorithm is designed for parallel execution on multiple machines, each hosting a subset of input data.

1.2 Outline

In the next chapter we explore the necessary background for the thesis. In Chapter 3 we elaborate our main contributions and also identify the delimitation of this thesis work. We conclude the work in Chapter 4. The complete publications, on which this thesis is based, are presented in Part II. Chapters 5, 6, and 7 describe our work on large-scale data dissemination. Our graph partitioning algorithms are presented in Chapters 8 and 9. Finally, Chapter 10 presents our solution for the coreference problem.
Chapter 2

Background

“Our knowledge can only be finite, while our ignorance must necessarily be infinite.”
- Karl Popper

Since all our algorithms make use of gossip-based protocols in one way or another, in this chapter we introduce gossiping protocols in the context of peer-to-peer networks, where they were first introduced. We explain how one can make use of gossiping protocols to provide a peer sampling service, a service that is utilized in most of our solutions. We also describe how a peer sampling service can be used to construct and maintain a desired topology. The topology construction is primarily important for our publish/subscribe solutions.

2.1 Peer-to-Peer Overlays

A peer-to-peer (P2P) overlay is an overlay network that exploits the existing resources at the edge of the network. Each node is represented by a peer, and plays the role of both client and server in the overlay network. Nodes cooperate to provide a distributed service, without the need for a single or centralized coordinator/server. The resources in such networks increase as more nodes join the network. Thus, peer-to-peer networks can potentially scale to a large number of participating nodes without having to dedicate powerful machines to provide the service. BitTorrent is a well-known example of such networks. In a peer-to-peer network nodes can join or leave the network continuously and concurrently. This phenomenon is called churn. Also network capacities change due to congestion, link failures, etc. Any such system, therefore, must handle churn in order to provide a reasonable quality of service.

Peer-to-peer overlays are mainly categorized into (i) structured, (ii) unstructured, and (iii) hybrid overlays. In a structured overlay, nodes acquire an identifier
from a globally known identifier space and are arranged to form a well defined topology. Such overlays should provide navigability, that is every node should be able to route to any other node in few, usually logarithmic, number of steps. This is achieved by utilizing a greedy distance-minimizing lookup service over the topology. Chord [3], Pastry [4], Kademlia [5], Symphony [6], CAN [7], One-hop DHT [8] and Oscar [9–11] are examples of the structured overlays.

One the other hand, unstructured overlays usually do not have a predefined topology and nodes randomly discover and select each other to link with. Lookup in these overlays usually takes the form of either flooding or random walk. Gnutella [12] and Kaza [13] are two examples of unstructured overlays.

While structured overlays are more efficient in routing, they need to be constantly maintained in the presence of churn in the network. On the other hand, unstructured overlays are very robust and automatically adapt to the changes in the network, though they can not guarantee a bounded routing time. Hybrid overlays exploit the best of the two worlds and are optimized for specific purposes. In such an overlay, some links are chosen with predefined criteria that lead to better routing performance, while some other links are selected randomly or based on other characteristics that are important for the application. For example, in our publish/subscribe systems, we construct hybrid overlays that are specially designed for connecting users with similar subscriptions together.

2.2 Gossiping Protocol

The basic gossiping protocol is based on a symmetric information exchange between pairs of nodes in a network. Each node has a local state, which is determined by the logic of the application. For instance, it could indicate the node’s load or cluster identifier. Whatever the state is, a node periodically shares it with its neighbors, i.e., the nodes that are directly connected to it. The basic protocol is illustrated in Algorithms 1 and 2. Every node $p$ has two different threads, one active and one passive. The active thread periodically initiates an information exchange with a random neighbor $q$, i.e., node $p$ sends a message containing its local state $S_p$ to node $q$. The neighbor selection policy that implements the function GetNeighbor() depends on the nature of the application. Node $p$ then waits for a response from $q$, and when the response is received, $p$ will update its current state. The passive thread at each node, for example node $p$, waits for messages sent by other nodes, and as soon as it receives a message, it will send back a reply containing its own state $S_p$, and also updates its state with the newly received value. Note, the information exchange is symmetric, since both participants send their states and receive each other’s state. When nodes have both states, they run the Update() method, which is again application dependent.

In the next section, we will see a few options for implementing this generic function for building peer sampling services (PSS). We use PSSs in our publish/subscribe systems (Papers A to C) as well as our graph partitioning algorithms...
2.3. PEER SAMPLING SERVICES

Algorithm 1 Generic Gossiping Protocol - Active thread at node p

1: procedure Gossip  
2: \( q \leftarrow \text{GetNeighbor}(\)  
3: \( \text{Send} \ S_p \text{ to} \ q \)  
4: \( \text{Recv} \ S_q \text{ from} \ q \)  
5: \( S_p \leftarrow \text{Update}(S_p, S_q) \)  
6: end procedure

Algorithm 2 Generic Gossiping Protocol - Passive thread at node p

1: procedure RespondToGossip  
2: \( \text{Recv} \ S_q \text{ from} \ q \)  
3: \( \text{Send} \ S_p \text{ to} \ q \)  
4: \( S_p \leftarrow \text{Update}(S_p, S_q) \)  
5: end procedure

(Papers D and E). For the latter algorithms, in addition to a PSS, we have another gossiping protocol, in which nodes’ states contains the partition they belong to as well as the partitions they neighbors belong to. These information is exchanged between pair of node and accordingly nodes update their state based in the received information. Moreover, we use gossiping in our diffusion-based community detection (Paper F), where the state of nodes is their color inventory and the cluster they belong to. In each round, nodes communicate not with a single neighbor, but with all their neighbors. The update method is also implemented such that it takes into accounts the incoming values from all neighbors. For more specific information about nodes states and the update function for each algorithm, please refer to the corresponding paper(s).

2.3 Peer Sampling Services

Peer sampling services (PSS) have been widely used in large scale distributed applications, such as information dissemination [14], aggregation [15], and overlay topology management [16–19]. The main purpose of a PSS is to provide the participating nodes with uniformly random sample of the nodes in the system. Gossiping algorithms are the most common approach to implementing a PSS [20–26]. In a gossip-based PSS, protocol execution at each node is divided into periodic cycles. In each cycle, every node selects a node from its partial view and exchanges a subset of its partial view with the selected node. Subsequently, both nodes update their partial views. Implementations of a PSS vary based on a number of different policies [21]:

1. **Node selection**: determines how a node selects another node to exchange information with. It can be either randomly (\textit{rand}), or based on the node’s age (\textit{tail}).
Algorithm 3 T-Man - Active Thread

1: procedure ExchangeRT
2:     neighbor ← selectRandomNeighbor()
3:     buffer ← getSampleNodes()
4:     buffer.merge(RT)
5:     Send [buffer] to neighbor
6:    Recv newBuffer from neighbor
7:     buffer.merge(newBuffer)
8:     RT ← selectNeighbors(buffer)
9: end procedure

Algorithm 4 T-Man - Passive Thread

1: procedure RespondToRTExchange
2:     Recv buffer from neighbor
3:     newBuffer ← getSampleNodes()
4:     newBuffer.merge(RT)
5:     Send [newBuffer] to neighbor
6:     newBuffer.merge(buffer)
7:     RT ← selectNeighbors(newBuffer)
8: end procedure

2. View propagation: determines how to exchange views with the selected node. A node can send its view with or without expecting a reply, called push-pull and push, respectively.

3. View selection: determines how a node updates its view after receiving the nodes’ descriptors from the other node. A node can either update its view randomly (blind), or keep the youngest nodes (healer), or replace the subset of nodes sent to the other node with the received descriptors (swapper).

In our work, we employ a light-weight peer sampling service, for providing each node with a uniformly random set of existing nodes in the system. Such service allows our systems to work without the need for any global knowledge at any point.

2.4 Topology Management

The overlay topology management is one of the applications that benefits from peer sampling services. In this thesis, we utilize T-man [16], which is a generic protocol for topology construction and management. In T-man, each node, \( p \), periodically exchanges its routing table (RT) with a neighbor, \( q \), chosen uniformly at random among the existing neighbors in the routing table. Node \( p \), then, merges its current routing table with \( q \)’s routing table, together with a fresh list of the nodes, provided by the underlying peer sampling service (Algorithms 3, lines 2-7). The resulting list becomes the candidate neighbors list for \( p \). Next, \( p \) selects a number of neighbors among the candidate neighbors and refreshes its current routing table. The same process will take place at node \( q \) (Algorithm 4). The core
2.5. SMALL-WORLD NETWORKS

The idea of our topology construction is captured in the neighbor selection mechanism, referred to as selectNeighbors in Algorithms 3 and 4. Such flexibility in neighbor selection makes it possible to construct any desirable topology, from a single ring or random graph, to any complex topology like torus, etc.

2.5 Small-world Networks

The small-world phenomenon refers to the property that any two individuals in a network are usually connected through a short chain of acquaintances. The existence of such chains have been long studied by researchers in different sciences, ranging from mathematics and physics to sociology and communication networks.

In 2000, Kleinberg [27] argued that there exist two fundamental components to this phenomenon. One is that such short chains are ubiquitous and the other is that individuals are able to find these short chains, using only local information. Kleinberg introduced the notion of distance and showed that in a small-world network two nodes are connected not uniformly at random, but with a probability that is inversely proportional to their distance. More precisely, nodes \( u \) and \( v \) are connected to one another with probability \( d(u, v)^{-\alpha} \), where \( d(u, v) \) denotes the distance between the two nodes, and \( \alpha \) is a structural parameter. Different values for \( \alpha \) yields a wide range of small-world networks, from random to regular graphs. However, Kleinberg mathematically proved that a greedy routing algorithms works best only if \( \alpha \) is equal to the number of dimensions in the network. In other words, navigation in a \( r \)-dimensional small-world network is most efficient only if nodes \( u \) and \( v \) are connected to each other with probability \( d(u, v)^{-r} \).

Many peer-to-peer systems, such as Symphony [6], Oscar [9, 11] and Mercury [28], have already used Kleinberg’s ideas to introduce overlay structures that are efficient in routing. We are also inspired by these works, in order to ensure a bounded routing complexity in our overlays.
Chapter 3

Thesis contribution

“Not everything that can be counted counts, and not everything that counts can be counted”
- William Bruce Cameron, Sociologist, 1963

In this thesis we show how a decentralized gossiping protocol can be used to (i) construct overlays that are efficient for information dissemination, (ii) partition big graphs into partitions of any given number or size, and (iii) find communities in a graph that we construct to solve the cross-document coreference problem. In the next sections, we list our detailed contributions, as well as the delimitation of our work, in each of the mentioned fields separately.

3.1 Information Dissemination

The amount of data in the digital world surrounding us is increasing very rapidly. According to a study by IBM, “15 petabytes of data are created every day - 8 times the volume housed in all US libraries” [29]. Thus, finding the relevant information is becoming more like looking for a needle in a haystack. Publish/subscribe systems, or pub/sub systems for short, leverage this problem by providing users with only the information they are actually interested in. Users of such systems utilize a subscription service to express their interest in specific data by either subscribing to a priori-known categories of data or defining filters over the content of the information they want to receive. These subscription models are called topic-based and content-based, respectively [30]. In both models, whenever some new data appears in the system, the interested subscribers are notified.

Currently, the majority of these systems use a client/server model and rely on dedicated machines to provide subscribe services. However, with a rapidly growing number of users on the Internet, and a highly increasing number of subscriptions, it is becoming necessary to use decentralized models for providing such a service at a reasonable cost. Moreover, the centralized model raises a privacy problem, since
all the user interests are revealed to a central authority, while in the real life most
users are reluctant to give away their personal interests for various privacy reasons.
Therefore, researchers have turned to peer-to-peer overlays, as an alternative design paradigm to the centralized model. Peer-to-peer overlays, if well implemented, exploit the resources at the edges of the network to provide a scalable service at a low or almost no cost. The available resources in a peer-to-peer network grow/shrink when more nodes join/leave the system. However, continuous joins and fails in such networks should be gracefully handled in order to provide a reasonable quality of service. Many peer-to-peer publish/subscribe systems have been proposed so far. However, they either

- require a potentially unbounded number of connections per node, which renders the system unscalable, or

- are potentially inefficient in routing, which results in large message delivery latencies, or

- put a heavy and/or unbalanced load on the nodes, which could ultimately lead to rapid deterioration of the system’s performance once the nodes start dropping the messages or choose to permanently abandon the system.

The main contributions on the subject of information dissemination are presented in form of two systems, *Vitis* [31, 32] and *Vinifera* [33], for topic-based and content-based publish/subscribe models, respectively. These contributions, which have been fully presented in Chapters 5, 6 and 7, can be summarized as follows:

- introducing novel algorithms for how to construct an overlay that adapts to user subscriptions and exploits the similarity of interests. With the use of gossip, we effectively cluster together the nodes with similar or overlapping subscriptions, while every node maintains only a bounded number of connections. These clusters are later exploited to reduce the amount of traffic overhead that is generated in the network.

- introducing a novel algorithm for leader election inside clusters by using only the undergoing gossiping protocol. These leaders, called *gateways* in Vitis terminology, are utilized to connect clusters of nodes with similar interest, while the generated traffic overhead is kept low.

- building efficient data dissemination paths over the clusters by enabling rendezvous routing over unstructured overlays. This is achieved by injecting structure into an otherwise unstructured overlay, using the ideas in the Kleinberg’s model. We guarantee that the event delivery time complexity is in logarithmic order.

- introducing load balancing mechanisms that adapt the overlay structure to the load of the published messages.
• combining multiple techniques from various fields, including gossiping, structured overlays and hashing techniques, to construct systems that outperform the existing state-of-the-art solutions.

• implementing and evaluating these systems in simulation, using both synthetically generated and real-world data traces.

Delimitations

• **Durability.** *Durability* refers to the property that a generated data item will survive in the system permanently. This property is of great importance for many applications, specially database systems. A publish/subscribe system can also be augmented by a durable storage system, which guarantees the persistency of events, as well as subscriptions. A lot of research is going on to design distributed storage systems and key-value stores which provide such guarantees. However, these works are orthogonal to our work and are considered out of the scope of this document.

• **Content filtering and matching techniques.** There are some interesting work on how to filter data content in the overlay networks [34–36]. However, these works are orthogonal and can be complementary to our solutions. In particular, we can utilize [34] on top of our dissemination trees in order to better filter out the published content.

• **Security attacks and byzantine behaviors.** Although security issues are practically important in real-world systems, the research work to address such issues are also orthogonal to our work and are considered out of the scope of this research. We assume all nodes behave in accordance with the protocols. However, node and link failures are handled in our systems.

### 3.2 Graph Partitioning

Finding good partitions is a well-known and well-studied problem in graph theory [37]. In its classical form, graph partitioning usually refers to *edge-cut* partitioning, that is, to divide vertices of a graph into disjoint clusters of nearly equal size, while the number of edges that span separated clusters is minimum. There are some studies [38–40], however, that show tools that utilize edge-cut partitioning do not achieve good performance on real-world graphs (which are mostly power-law graphs), mainly due to unbalanced number of edges in each cluster. In contrast, both theory [41] and practice [42, 43] prove that power-law graphs can be efficiently processed in parallel if *vertex-cuts* are used. In contrast to edge-cut partitioning, a vertex-cut partitioning divides edges of a graph into equal size clusters. The vertices that hold the endpoints of an edge are also placed in the same cluster as the edge itself. However, the vertices are not unique across clusters and might have to
be replicated (cut), due to the distribution of their edges across different clusters. A good vertex-cut is one that requires minimum number of replicas. Figure 3.1 illustrate the difference between these two types of partitioning.

We focus on processing extremely large-scale graphs, e.g., user relationship and interaction graphs from on-line social networking services such as Facebook or Twitter, resulting in graphs with billions of vertices and hundreds of billions of edges. The very large scale of the graphs we target poses a major challenge. Although a very large number of algorithms are known for graph partitioning [44–51], including parallel ones, most of the techniques involved assume a form of cheap random access to the entire graph. In contrast to this, large scale graphs do not fit into the main memory of a single computer, in fact, they often do not fit on a single local file system either. Worse still, the graph can be fully distributed as well, with only very few vertices hosted on a single computer.

We provide a distributed balanced graph partitioning algorithm, called JA-BE-JA, both for edge-cut and vertex-cut partitioning. Choosing between edge-cut and vertex-partitioning depends on the application, and JA-BE-JA, to the best of our knowledge, is the only algorithm that can be applied in both models. JA-BE-JA is a decentralized local search algorithm and it does not require any global knowledge of the graph topology. That is, we do not have cheap access to the entire graph and we have to process it only with partial information. Each vertex of the graph is a processing unit, with local information about its neighboring vertices, and a small subset of random vertices in the graph, which it acquires by purely local interactions. Initially, every vertex/edge is assigned to a random partition, and over time vertices communicate and improve upon the initial assignment.

Our algorithm is uniquely designed to deal with extremely large distributed graphs. The algorithm achieves this through its locality, simplicity and lack of synchronization requirements, which enables it to be adapted easily to graph processing frameworks such as Pregel [2] or GraphLab [1]. Furthermore, JA-BE-JA can be applied on fully distributed graphs, where each network node represents a single graph vertex.

To evaluate JA-BE-JA for edge-cut partitioning, we use multiple datasets of different characteristics, including a few synthetically generated graphs, some graphs that are well-known in the graph partitioning community [52], and some sampled graphs from Facebook [53] and Twitter [54]. We first investigate the impact of different heuristics on the resulting partitioning of the input graphs, and then com-
3.3. COREFERENCE RESOLUTION

pare JA-BE-JA to METIS [45], a well-known centralized solution. We show that, although JA-BE-JA does not have cheap random access to the graph data, it can work as good as, and sometimes even better than, a centralized solution. In particular, for large graphs that represent real-world social network structures, such as Facebook and Twitter, JA-BE-JA outperforms METIS [45].

For vertex-cut partitioning, we will compare our solution with a state-of-the-art system [55], and we show that JA-BE-JA not only guarantees to keep the size of the partitions balanced, but also outperforms its counterparts with respect to vertex-cut.

The main contributions on this subject are published in two papers, JA-BE-JA [56] and JA-BE-JA-vc [57], for edge-cut and vertex-cut partitioning, respectively. These papers are fully presented in Chapters 8 and 9.

**Delimitations**

- Edge-cut partitioning for the graphs with weighted vertices can not be readily addressed with JA-BE-JA. The reason is JA-BE-JA requires the initial color distribution to be an invariant. It is for exactly this reason, that graph vertices can not change their colors independently and have to find some other vertices to swap their color with. However, this will only work if the two swapping vertices are of equal weights. If vertices have different weights, the color distribution before and after the swap will not be the same. Likewise, JA-BE-JA-vc is not readily applicable to vertex-cut partitioning for graphs with weighted edges.

3.3 Coreference Resolution

Resolving entities in a text may not always be a difficult task for humans. When one comes across Mercury in an article about the solar system, they instantly think of Mercury, the planet, and not about Mercury, the chemical element or Freddie Mercury. For a computer though, such a disambiguation requires a considerable amount of processing. This problem, i.e., the task of disambiguating manifestations of real world entities in various records or mentions, is known as Entity Resolution or Coreference Resolution. The ambiguity arises from the fact that the same word can refer to multiple entities. Often disambiguation is required across multiple documents. For example, there are various articles across the web that contain news about Mercury. Given a set of such documents with an ambiguous mention (Mercury, for example), the Cross-Document Coreference problem seeks to group together those documents that talk about the same entity in real world (e.g., one group for the planet, one for the chemical element, etc.).

This problem is challenging because: (i) often the number of underlying entities and their identities are not known (e.g., we do not know how many different Mercuries are to be discovered), and (ii) the number of possible classifications grows exponentially with the number of input documents.
A widely used approach to this problem, known as Mention-Pair model, is to compute a pair-wise similarity value based on the common keywords that exist in each pair of documents [58]. If two documents are found similar more than a pre-defined threshold, they are classified together. However, this requires huge amount of computations. The high complexity of the Mention-Pair model renders it impractical for web-scale coreference, where we have to process millions of documents in a reasonable time. Even after the costly computation step, a clustering step is required to partition the mentions into coreferent groups. The clustering itself is a challenging task and is known to be NP-hard.

We propose a novel approach to coreference resolution, which does not require separate classification and clustering steps. Instead, we transform the problem to a vertex-centric graph processing task. This enables us to take advantage of the recent advances in graph processing frameworks, such as GraphChi [59] or GraphLab [1], and apply our algorithm to extremely large graphs.

To construct the graph, we create two types of nodes. One type represents the ambiguous word, which we assume is given in advance. Another type of nodes represents the unambiguous words that surround the ambiguous word in each document. Since we do not know whether or not different mentions of the ambiguous word are referring to the same real-world entity, we create as many nodes as the number of documents mentioning them. The unambiguous words might as well appear in multiple documents. For them, however, we do not create a new node, if they already exist. Finally, we add an edge between two nodes, if their corresponding words co-occurred in the same document. Consequently, each single document is represented by a full mesh, or clique, of all its keywords.

We will then observe that some cliques overlap, which indicates that their corresponding documents have a similar context. In fact, the main insight to our work is that the topological community structure of the constructed graph identifies similar contexts and thus, is an accurate indicator of the coreference classification. Based on this fact, we propose a novel community detection algorithm for coreference resolution. Our algorithm is diffusion based and exploits the fundamentals of flow networks. In such a network each node has a capacity and each edge can transfer a flow, just like a pipe, between two nodes. We envision multiple flows in our graph, one per community. To distinguish these flows, we assign a distinct color to each of them.

Initially each single document constitutes a distinct community, i.e., it will be assigned to a unique color. All the nodes that belong to a document will get a unit of the color of their document. Therefore, those nodes that are shared between documents, will receive multiple units of colors. However, each node always identifies itself with only a single color, which has the highest collective volume in its neighborhood, so-called the dominant color. Nodes continuously exchange parts of their colors with their neighbors by diffusing the colors through their links. Therefore, the available volume of color at nodes, and accordingly the dominant color in their vicinity, changes during the course of algorithm. We will show that with appropriate diffusion policies it is possible to accumulate one distinct color
3.3. COREFERENCE RESOLUTION

in each of the well connected regions of the graph. Finally, the ambiguous nodes that end up having the same dominant color are considered to be coreferent. Since our constructed graph is sparse, the overhead of such computation remains low. Moreover, we can produce more accurate results, compared to the state of the art. This twofold gain is owed to the combination of two ideas, that constitute our main contributions:

- a technique for transforming the expensive coreference problem, into a graph problem, in which the coreferent words belong to the same topological community structure. The graph that we construct is sparse, because those documents that have dissimilar contexts, will have very few or even no direct connections. The computations on the graph are performed per edge basis, i.e., only if there is an edge between two nodes, they will communicate some flows. Hence, the irrelevant documents which are weakly connected, if not disconnected, will not impose any computation in the graph. At the same time, a more thorough search of the solution space is possible, as we are not limited to pair-wise similarity discoveries only. Instead, similarity between any number of documents is naturally captured within the community structures that emerge from the inter-linked context words.

- a novel node-centric diffusion-based community detection algorithm that mainly uses local knowledge of the graph at each node. Hence, it allows for highly parallel computations and usage of the existing graph processing frameworks.

We run our algorithm on different datasets, which are transformed to graphs with distinct structural properties. For example, on a baseline dataset for person name disambiguation, we produce a classification with an F-score 15% higher than that of the state of the art algorithm by Singh et al. [60]. Moreover, on a dataset provided in the Word Sense Induction task of SemEval 2010, we achieved as good F-score as the best reported result. However, we considerably outperform the other solutions with respect to a complementary accuracy metric, which measures the average number of items in each clusters. The full paper is presented in Chapter 10.

Delimitations

- To discover which word(s) is (are) ambiguous, is a different problem which is orthogonal to our work and out of the scope of this document.

- Tagging the classified entities, i.e., annotating the entities with the correct label requires external knowledge, for example from Wikipedia or DBpedia, and is orthogonal to our work, hence, out of the scope of this document.
3.4 Publications


List of the publications that are not included in this thesis.


Chapter 4

Conclusions

“Success is not final, failure is not fatal: it is the courage to continue that counts.”
- Winston Churchill

We believe that without parallel and distributed algorithms, big data processing will become prohibitively costly and the new advances in the emerging computing frameworks will not be effectively utilized. Hence, developing decentralized algorithms that can operate on partial data on top of these computing frameworks is absolutely necessary. Our work is a small step in that direction.

We employed the power of gossip to solve various problems, including data dissemination, partitioning and disambiguation. Gossiping enables all our solutions to work with partial knowledge and with a very low communication overhead. Thus, our algorithms can be executed efficiently and effectively in a distributed environment.

Our solutions for data dissemination scale to very large networks, while producing a relatively low overhead. Data is delivered to the interested users (subscribers) reliably and fast. Even in the presence of failures the delivery rate of the system remains high. In particular, we presented Vitis, a topic-based publish/subscribe system, which scales with the number of nodes as well as the number of topics in the overlay. The main contribution of this work is a novel hybrid publish/subscribe overlay that exploits two ostensibly opposite mechanisms: unstructured clustering of similar nodes and structured rendezvous routing. We employed a gossiping technique to embed a navigable small-world network, which efficiently establishes connectivity among clusters of nodes that exhibit similar subscriptions. We also give a theoretical bound on the worst case delay. Moreover, we showed how we can embed locality information into the overlay topology, that is, how to make the connections between nodes to better reflect the physical network connections. We introduced a notion of distance in the neighbor selection mechanism of Vitis and studied how we can exploit locality-awareness in-line with the subscription correlations. Finally, we showed that Vitis adapts to biased rates of events that are
published on different topics, and builds more efficient groups for hot topics, thus, improving the overall performance of the event dissemination.

We also presented Vinifera, a content-based publish/subscribe system that uses a gossip-based technique to construct a topology that not only resembles a small-world network, but also connects the nodes with similar subscriptions together. On top of this hybrid overlay, we utilized a rendezvous routing mechanism to propagate node subscriptions in the overlay. Together with an order preserving hashing technique and an efficient showering algorithm we enabled range queries, and at the same time, we employed a load balancing technique to deal with the potential non-uniform user subscriptions. The combination of all these techniques are seamlessly integrated within a single gossiping layer, thus keeping Vinifera simple, lightweight and robust.

For graph partitioning, we provided JA-BE-JA, an algorithm that to the best of our knowledge, is the first distributed algorithm for balanced edge-cut partitioning that does not require any global knowledge. To compute the partitioning, nodes of the graph require only some local information and perform only local operations. Therefore, the entire graph does not need to be loaded into memory, and the algorithm can run in parallel on as many computers as available. We showed that our algorithm can achieve a quality partitioning, as good as a centralized algorithm. We also studied the trade-off between the quality of the partitioning versus the cost of it, in terms of the number of swaps during the run-time of the algorithm.

We also presented JA-BE-JA-VC, a distributed and parallel algorithm for vertex-cut partitioning. JA-BE-JA-VC partitions edges of a graph into a given number of clusters with any desired size distribution, while the number of vertices that have to be replicated across clusters is low. In particular, it can create balanced partitions while reducing the vertex-cut. JA-BE-JA-VC is a local search algorithm that iteratively improves upon an initial random assignment of edges to partitions. It also utilizes simulated annealing to prevent getting stuck in local optima. We compared JA-BE-JA-VC with two state-of-the-art systems, and showed that JA-BE-JA-VC not only guarantees to keep the size of the partitions balanced, but also outperforms its counterparts with respect to vertex-cut.

Finally, we presented a graph-based approach to coreference resolution. We showed that by using a graph representation of the documents and their context, and applying a community detection algorithm we can speed up the task of coreference resolution by a very large degree. The accuracy of coreference resolution could also be improved at the same time, because we are able to search beyond only pair-wise comparisons. The graph that we construct enables us to discover any existing closeness/similarity between any subset of documents. Thus, we can explore the solution space more freely and more smartly.
Part II

Research Papers
Chapter 5

Vitis: A Gossip-based Hybrid Overlay for Internet-scale Publish/Subscribe
Enabling Rendezvous Routing in Unstructured Overlay Networks

Fatemeh Rahimian, Amir H. Payberah, Sarunas Girdzijauskas and Seif Haridi

In the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Anchorage, Alaska, USA, May 2011.
5.1. INTRODUCTION

Peer-to-peer overlay networks are attractive solutions for building Internet-scale publish/subscribe systems. However, scalability comes with a cost: a message published on a certain topic often needs to traverse a large number of uninterested (unsubscribed) nodes before reaching all its subscribers. This might sharply increase resource consumption for such relay nodes (in terms of bandwidth transmission cost, CPU, etc) and could ultimately lead to rapid deterioration of the system’s performance once the relay nodes start dropping the messages or choose to permanently abandon the system. In this paper, we introduce Vitis, a gossip-based publish/subscribe system that significantly decreases the number of relay messages, and scales to an unbounded number of nodes and topics. This is achieved by the novel approach of enabling rendezvous routing on unstructured overlays. We construct a hybrid system by injecting structure into an otherwise unstructured network. The resulting structure resembles a navigable small-world network, which spans along clusters of nodes that have similar subscriptions. The properties of such an overlay make it an ideal platform for efficient data dissemination in large-scale systems. We perform extensive simulations and evaluate Vitis by comparing its performance against two base-line publish/subscribe systems: one that is oblivious to node subscriptions, and another that exploits the subscription similarities. Our measurements show that Vitis significantly outperforms the base-line solutions on various subscription and churn scenarios, from both synthetic models and real-world traces.

5.1 Introduction

Publish/subscribe systems are nowadays widely used over the Internet. News syndication (RSS feeds), multi-player games, social networks such as Twitter or Facebook, media streaming applications, e.g., Spotify, or IPTV, are a few examples of such systems. Users of these systems express their interest in certain data, by subscribing to a number of topics, which can be daily news, a friend’s tweets, a music playlist, or a channel on IPTV. Should any new data be published on a topic, the
subscribers are notified and provided with the content. Depending on the application, this service could be bandwidth intensive and/or time critical, as in live streaming applications, or may include a large number of topics, as it is the case for Spotify playlists or social networks.

Currently, the majority of these systems use a client/server model and rely on dedicated machines to provide subscribe services. However, with a rapidly growing number of users on the Internet, and a highly increasing number of topics, it is becoming necessary to use decentralized models for providing such a service at a reasonable cost. Therefore, a lot of work has been done to design peer-to-peer publish/subscribe systems.

To provide the subscribe service, a range of solutions has been proposed. On the one extreme, nodes construct a separate overlay per topic, i.e., each node becomes a member of as many overlays as the number its subscriptions (e.g., Rappel [61] or Tera [62]). Although a node in these systems only receives the events that it has subscribed for, the number of the node’s connections and, therefore, the overlay management cost grow linearly with the number of topics the node subscribes to. This, potentially, renders the system unscalable, when nodes subscribe to very large number of topics, e.g., thousands of topics, as is the case in some real world applications, e.g., [63].

At the other extreme, nodes use a bounded number of connections to manage all their subscriptions simultaneously (e.g., Scribe [64] or Bayeux [65]). These solutions, however, suffer from high traffic overhead, that is, nodes have to send and receive data in which they have no interest. As we will demonstrate later, in order to make sure all the subscribers of a topic receive their intended data, many nodes that are not subscribed for that topic have to get involved in data dissemination. Note that, although peer-to-peer users are generally willing to contribute their resources to the system, they might lose incentive to cooperate, if the amount of traffic they forward exceeds their expectations. For example, a user of IPTV might permanently leave the overlay if it has to constantly forward a large media stream in which it has no interest. Therefore, it is crucial to decrease the traffic overhead of the nodes.
In this paper, we introduce Vitis, a topic-based publish/subscribe solution, that fills in the gap between the aforementioned extremes, while taking the best properties from both sides: bounded node degree and low traffic overhead. Vitis nodes run a gossip-based peer sampling service [20], to exploit the subscription similarities and select as neighbors, nodes with whom they share the most topics. Without the limitation on the node degree, a separate overlay per topic could eventually be formed. However, due to the bounded node degree, there is no guarantee that all the nodes, which are interested in a topic, connect together. In fact, any number of clusters for the same topic can emerge in different parts of the overlay (Figure 5.1).

We denote a cluster for a topic as a maximal connected subgraph of the overlay, which includes a set of nodes that are all interested in that topic. Nodes inside a cluster are reachable from one another. In order to make sure a published event for a topic is delivered to all the subscribers, all the clusters of that topic must be linked together via other nodes. The path that connects different clusters of the same topic is called relay path. Such a path, includes nodes that are not interested in the topic themselves. We refer to these nodes as relay nodes, hereafter. The challenge is to decrease the number of required relay nodes, while making sure that all the clusters associated with a topic (and therefore, all the nodes interested in that topic) are linked together.

To enable relaying between the clusters, we introduce a novel technique for rendezvous routing [66] on top of an unstructured overlay. For that, Vitis nodes...
form a navigable small-world overlay (Figure 5.2), which is shown to have the best decentralized routing performance [67]. Then, nodes in each cluster select a number of representative nodes, as gateways. The number of gateways for a cluster is proportional to the diameter of the subgraph that represents the cluster. Gateway nodes are responsible for employing the navigable small-world overlay to connect to other clusters for the same topic. They perform a greedy lookup for the topic id, and all meet at the same node, i.e., rendezvous node. This approach is comparable to Scribe or Bayeux, but the difference is that nodes are efficiently grouped together in advance, and instead of each node independently performing the rendezvous routing, only few nodes, i.e., gateway nodes, establish the relay paths. In section 5.3 we elaborate on how the gateway nodes are selected and how the relay paths are established. We also show that the event propagation delay, in terms of the number of hops, is bounded to $O(\log^2 N)$, in our system. The resulting structure resembles a grapevine, with clusters of grape hanging from the canes, thus, inspired the name Vitis.

We evaluated the performance of Vitis through extensive large-scale simulations, with synthetic data as well as real-world subscription traces from Twitter [54], and churn traces from Skype [68]. We compare our system, with two base-line solutions: (i) a rendezvous routing system which is based on a structured overlay, with a bounded node degree, and oblivious to node subscriptions, and (ii) an unstructured solution that exploits the subscription correlation between nodes, without any bound on node degree. The results show that the traffic overhead in Vitis is between 40% to 75% less than the first base-line solution. We also show that, with a bounded node degree, Vitis always deliver the events to all the subscribers, while the hit ratio degrades in the second base-line solution, when the node degree can not grow indefinitely.

In the next section we describe the related work and position Vitis in the field. In section 5.3 we go into the details of the solution and in section 5.4 we present the results of our experiments. We conclude the work in section 5.5.

### 5.2 Related Work

The traditional architectures for publish/subscribe systems are the client-server and broker-based models. In systems based on either of these models, the subscriptions are submitted to a server (or broker). Also publishers send their events to this server (or broker), where the events are matched to the user subscriptions and forwarded to the users, accordingly. Solutions such as Siena [69], Gryphon [70], Hermes [71] or Corona [72] are in this category.

A more recent architecture for designing publish/subscribe systems, replaces the client-server or broker-based models with peer-to-peer overlays. This enables Internet-scale applications with many users as well as many topics. The peer-to-peer overlays can be roughly classified into two main categories: structured and unstructured. Solutions such as Scribe [64] and Bayeux [65] are examples of struc-
5.2. RELATED WORK

tured overlay networks, while Tera [62], Rappel [61], StAN [73] and SpiderCast [74] fall into the second category, where a gossip-based approach is utilized. There are also solutions, like Quasar [75] or our solution, Vitis, which use gossiping to construct a hybrid of structured and unstructured overlays for event dissemination.

Regardless of how the overlay is constructed, the main challenge is to guarantee that nodes will receive all the events they have subscribed for, while not being overloaded with a large number of connections or excessive overhead. Tera [62], Rappel [61], StAN [73], and SpiderCast [74] construct a separate overlay for each topic. When a node subscribes to a topic, it becomes a member of that topic overlay. Therefore, published events for that topic are only distributed among the subscriber nodes and the traffic overhead is eliminated. However, nodes should join as many overlays as the number of topics they subscribe to. Thus, the node degree and overlay maintenance overhead grow linearly with the number of node subscriptions. This is, however, impractical for Internet-scale applications, when users subscribe to a large number of topics. We address this problem in Vitis, as nodes maintain a bounded number of connections, regardless of the number of their subscriptions.

To mitigate the scalability problem, SpiderCast [74] takes advantage of the similarity of interest between different nodes. The authors of SpiderCast argue that due to user subscription correlations, a single link can connect a node to more than one topic overlay. Thus, the number of required connections per node decreases. Since the user subscriptions are shown to be typically correlated in the real-world traces [76, 77], this idea works nicely with a limited number of node subscriptions. Nevertheless, the performance and scalability of SpiderCast is unknown, when the number of subscriptions is large or when there is churn in the environment. Moreover, any node in SpiderCast needs to have prior knowledge of at least 5% of other nodes in the system. In contrast, Vitis nodes do not need such a linear-scale amount of information about the other nodes in the system, and can subscribe to unbounded number of topics. In Section 5.4, we compare a SpiderCast-like system with Vitis and show that SpiderCast nodes suffer from maintaining a large number of connections, in order to receive all the events they have subscribed for.

There are also solutions that account for scalability by bounding the number of required connections per node, for example Quasar [75], which is a gossip-based solution, or Scribe [64] and Bayeux [65], which are DHT-based. In Quasar [75], each node exchanges with its nearby neighbors, an aggregated form of subscription information of itself and its neighbors a few hops away. Therefore, a gradient of group members for each topic emerges in the overlay. When a node publishes an event, targeted for a group, it sends multiple copies of the event in random directions along the overlay, and the event is probabilistically routed towards the group members. Quasar obviates the need for an overlay structure that encodes group membership information. However, it is inherently a probabilistic design model, even in a static environment. It also incurs high traffic overhead, since it is oblivious to nodes’ subscriptions and involves many uninterested nodes in the
event dissemination. In Vitis, on the other hand, we reach a full hit ratio, while minimizing the traffic overhead by organizing similar nodes into clusters.

In Scribe [64] or Bayeux [65], nodes are organized into a Distributed Hash Table (Pastry [69] and Tapestry [78], respectively), where each node maintains $O(\log N)$ connections. Then, a spanning tree is built for each topic, with a rendezvous node at the root, which delivers the events to the nodes that join the tree. This approach, however, forces many nodes to relay the events for which they have not subscribed, as they happen to be on the path towards the rendezvous node. Consequently, such systems suffer from a huge amount of traffic overhead. Vitis nodes also have a bounded node degree and form a tree-like structure per topic. However, unlike Scribe or Bayeux, the leaves in these trees are not single nodes, but groups of nodes, which are subscribed for that topic. We show through simulations, that an efficient clustering of nodes with similar interests, results in trees with far less intermediary nodes, and hence, much smaller traffic overhead.

Another solution, Magnet [79], exploits similar ideas of subscription correlation between the nodes, under the bounded node degree assumption. However, Magnet is purely based on a structured overlay and cannot fully capture the correlation between subscriptions, for it is bounded to one dimensional space, where the structured overlay is constructed. Also, Magnet is less robust in volatile environments, such as the Internet. In contrast, Vitis is not restricted to any dimension while capturing the subscription correlation (since clustering is done in an unstructured way) and as we show in our experiments, it is very robust due to the underlying gossip protocol.

Finally, there is recent work for resource location in clouds [80], which can be interpreted as a publish/subscribe system, though with quite clear differences. In [80], nodes query for a resource with certain attributes, and are redirected to a part of the cloud that contains the resources with requested properties. This work also employs a peer sampling service to build a structured and an unstructured overlay. In the unstructured overlay, resources with similar attributes are placed close to one another. However, [80] does not guarantee, and in fact does not need, that all the nodes with the queried properties are found. Nevertheless, in Vitis, we make sure that all the subscribers are found and informed of the published event. Moreover, [80] is not applicable for event dissemination, for it enforces a significant load on the nodes in the structured overlay.

### 5.3 Vitis

At a high level, Vitis borrows ideas from gossip based sampling services [20] (Section 5.3) and rendezvous routing on structured overlays [66]. While benefiting from these ideas, Vitis employs a technique for selecting nodes that share topic interests (Section 5.3), and introduces a novel way of constructing a dissemination structure that minimizes the traffic overhead in the network (Section 5.3).

Every Vitis node maintains a bounded-size *routing table (RT)*, which is a partial
list of the existing nodes in the system that the node uses for routing the messages. The entries in the routing table are selected either as (i) small-world connections, or (ii) similarity connections based on a preference function. Hereafter, we refer to these two type of connections as sw-neighbor and friends, respectively. We also use the term neighbor to refer to any of the entries in the routing table, either friend or sw-neighbor.

Moreover, each node has a profile, which includes a unique node id, and the id of topics that the node subscribes to. Node ids and topic ids share the same identifier space and are generated by a globally known hash function that generates ids that are uniformly distributed in the identifier space, e.g, SHA-1. The topic id for topic $t$ is denoted by $hash(t)$, hereafter. Subscribing to or unsubscribing from a topic, is done by adding or removing the topic id to/from the profile.

Every node periodically sends its profile to the nodes in its routing table, to inform them of its own subscriptions. This profile message also serves as a heartbeat message, and helps the nodes to constantly maintain their routing tables. When a node fails or leaves, its neighbors will stop receiving heartbeat messages and consequently, its entry will be removed from the routing table of its neighbors.

**Neighbor Selection**

Vitis utilizes a gossip-based peer sampling service to build a hybrid overlay. Any of the existing implementations for this service, e.g., [20, 21, 26, 81], can be used. When a node joins the overlay (Algorithm 5), it contacts a bootstrap node and receives a number of nodes to start communicating with. Then, the node runs the peer sampling service and periodically acquires fresh random samples of the existing nodes.

The overlay construction mechanism in Vitis is inspired by T-man [16], which is a generic protocol for topology construction and management. Each node, $p$, periodically exchanges its routing table (RT) with a neighbor, $q$, chosen uniformly at random among the existing neighbors in the routing table. Node $p$, then, merges its current routing table with $q$’s routing table, together with a fresh list of the nodes, provided by the underlying peer sampling service (Algorithms 6, lines 2-7). The resulting list becomes the candidate neighbors list for $p$. Next, $p$ selects a number of neighbors among the candidate neighbors and refreshes its current routing table. The same process will take place at node $q$ (Algorithm 7). The core idea of our topology construction is captured in the neighbor selection mechanism, referred to as selectNeighbors in Algorithms 6 and 7 and described in Algorithm 8.

As mentioned previously, the routing table includes sw-neighbors and friend links. We define a system parameter $k$ in Vitis, which determines the number of sw-neighbors in the routing table. The lower $k$ is, the higher the upper bound on the routing cost is [6], while nodes are better grouped together and the traffic overhead decreases. That is, there is trade-off between the traffic overhead and the propagation delay, which can be controlled by $k$. In Section 5.4 we investigate the impacts of this trade-off on the performance of the system.
Algorithm 5 Join

1: procedure Join
2: InitProfile()  \(\triangleright\) subscribe to topics
3: InitRoutingTable()  \(\triangleright\) get some neighbors from the bootstrap node
4: start PeerSamplingService()
5: do every \(\delta t\)  \(\triangleright\) repeat periodically
6: ExchangeRT()
7: ExchangeProfile()
8: end procedure

Algorithm 6 T-Man - Active Thread

1: procedure ExchangeRT
2: neighbor \(\leftarrow\) selectRandomNeighbor()
3: buffer \(\leftarrow\) getSampleNodes()  \(\triangleright\) provided by the peer sampling service
4: buffer.merge(RT)  \(\triangleright\) RT is the local routing table
5: Send [buffer] to neighbor
6:Recv newBuffer from neighbor
7: buffer.merge(newBuffer)
8: RT \(\leftarrow\) selectNeighbors(buffer)
9: end procedure

Algorithm 7 T-Man - Passive Thread

1: procedure RespondToRTExchange
2: Recv buffer from neighbor
3: newBuffer \(\leftarrow\) getSampleNodes()
4: newBuffer.merge(RT)
5: Send [newBuffer] to neighbor
6: newBuffer.merge(buffer)
7: RT \(\leftarrow\) selectNeighbors(newBuffer)
8: end procedure

Sw-neighbor selection

In order to perform rendezvous routing \cite{66}, Vitis nodes establish sw-neighbors by utilizing a mechanism similar to Symphony \cite{6}. Similar to Symphony, Vitis constructs a navigable small-world overlay, which guarantees a bounded routing cost that depends on the node degree. It introduces a distance function in the identifier space, where a neighbor for a node is selected with a probability that is inversely proportional to the distance between the two nodes.

The authors in \cite{6} showed that selecting \(k\) links according to this probability function, results in a routing cost of the order \(O\left(\frac{1}{k} \log^2 N\right)\) messages. For example, if one such neighbor is selected (as in Algorithm 8, line 8), the routing time is bounded to \(O\left(\log^2 N\right)\). Note that, unlike Symphony, in Vitis nodes establish their sw-neighbors via periodic gossiping.

Moreover, our gossip protocol (Algorithms 6 and 7) enables Vitis nodes to form a ring topology in the identifier space. The ring is required for lookup consistency.
Algorithm 8 Select Neighbors

```
1: procedure SelectNeighbors(buffer)
2:   successor ← findSuccessor(buffer)
3:   buffer.remove(successor)
4:   selectedNeighbors.add(successor)
5:   predecessor ← findPredecessor(buffer)
6:   buffer.remove(predecessor)
7:   selectedNeighbors.add(predecessor)
8:   sw-neighbor ← buffer.select-sw-neighbor(RANDOM-DISTANCE)
9:   buffer.remove(sw-neighbor)
10:  selectedNeighbors.add(sw-neighbor)
11:  for all node in buffer do
12:     utility[node] ← calculateUtility(node, self)
13:  end for
14:  sortedNeighbors ← utility[].sort()
15:  friends ← sortedNeighbors.top(RT-SIZE - 3)
16:  selectedNeighbors.add(friends)
17:  return selectedNeighbors
18: end procedure
```

in the overlay, which is, in turn, required for constructing the relay paths (See Section 5.3). Therefore, two entries of the routing tables are always dedicated for maintaining the neighbors on the ring. Each node selects two nodes with the closest id to its own, in the two directions, among the nodes it has learnt about so far, as its predecessor and successor on the ring (Algorithm 8, lines 2 and 6). Although initially the predecessors and successors may not be correctly assigned, T-Man protocol guarantees that through periodic gossiping the ring topology rapidly converges to a correct ring and is constantly maintained, thereafter [16].

Friend selection

The remaining candidate neighbors are ordered by a preference function. A node, then, selects the highest ranked nodes from this list (Algorithm 8, lines 11-15). The preference function takes into account: (i) the interest similarity of the nodes, as well as (ii) the event publication rate for different topics. It can also be extended to account for the underlying network topology and reduce the cost of data transfer in the physical network. The preference function, gives a pair-wise utility value to the nodes, according to the following function:

\[
utility(i, j) = \frac{\sum_{t \in subs(i) \cap subs(j)} rate(t)}{\sum_{t \in subs(i) \cup subs(j)} rate(t)} \quad (5.1)
\]

where \(subs(i)\) indicates the set of topics that node \(i\) has subscribed to, and \(rate(t)\) is the publication rate of topic \(t\).
If the distribution of published events on different topics is uniform, nodes that have bigger interest overlap relative to the total number of their subscriptions, end up as friends. For example, if node $p$ subscribes to topics $\{A,B,C\}$, node $q$ subscribes to $\{C,D\}$, and node $r$ subscribes to $\{C,D,E,F,G,H\}$, then $utility(p,q) = 0.25$, $utility(p,r) = 0.125$, and $utility(q,r) = 0.33$. That means, node $p$ will prefer $q$ to $r$, although it shares exactly one topic with both of them. Thus, node $p$ less probably gets involved in the event propagation of events on topics $\{E,F,G,H\}$, in which it has no interest. Likewise, nodes $q$ and $r$ prefer to keep $r$ and $q$ in their local views, respectively.

If the publication rate varies for different topics, the interest overlaps are weighted by the publication rates. For example, if the publication rate for topic $t$ goes to zero, i.e., almost no event is published on $t$, then $t$ is practically ignored in the preference function. On the other hand, nodes will give a high utility to one another, if they are interested in a common topic that has a high rate of events.

**Relay Path Construction**

As we explained in Section 5.3, the routing table size is bounded, thus, not all neighbors with utility greater than zero will be selected. As a result, instead of a unique cluster per topic, multiple disjoint clusters can emerge in the overlay. A cluster for topic $t$, is a maximally connected subgraph of the nodes that are all interested in $t$. If topic $t$ has $n$ disjoint clusters, these clusters are numbered and denoted as $C_i[t]$, where $i$ is from 1 to $n$. To ensure that all $n$ clusters of topic $t$ are connected, some other nodes that are not subscribed to $t$ have to get involved.

We define a *rendezvous node* for topic $t$, as a node with the closest id to $hash(t)$. Since Vitis constructs a small-world overlay, any node is able to route to any other node in the identifier space. To find the rendezvous node, a node performs a lookup on $hash(t)$, and all the nodes on the lookup path become *relay nodes* for $t$. This path, which we refer to as relay path, can include any kinds of links, e.g., friend, sw-neighbor or ring links (Figure 5.3). This is equivalent to the concept explored in Scribe [64] or Bayeux [65], where nodes subscribe on the path towards the rendezvous node and ultimately build a spanning tree.

In order to minimize the number of relay nodes for a topic, instead of letting each subscriber node route to the rendezvous node, as in, e.g., Scribe, nodes inside each cluster select a number of representative nodes, as gateways, to establish the relay path.

Algorithm 9 defines the gateway selection process. To select a gateway for cluster $C_i[t]$, each node in $C_i[t]$ initially proposes itself as gateway (Algorithm 9, line 3). This proposal is piggybacked on the node profile that is periodically sent to the neighbors (Algorithm 10). Likewise, the node receives other proposals from its neighbors, and revises its proposal for the next round (Algorithm 9, line 19). To avoid loops, each proposal also includes the node which proposed the gateway. This node is denoted as parent in Algorithm 9. Among the proposed gateways, the node selects as gateway the one that has the closest id to $hash(t)$, measured
Algorithm 9 Update Profile

1: procedure UpdateProfile
2:   for all topic in profile.subscriptions do
3:     prop ← initProposal(self, self, 0)     \>
4:     for all neighbor in RT do
5:       if neighbor.isInterested(topic) then
6:         new ← neighbor.getProposal(topic)
7:         if new.parent ≠ new.parent OR new.parent ∉ RT then
8:           newDis = distance(new.GW, hash(topic))
9:         if newDis < currentDis AND new.hops + 1 < d then
10:            prop ← (new.GW, neighbor, new.hops + 1)
11:       end if
12:     end if
13:   end if
14: end for
15: profile.subscriptions.update(topic, prop)
16: if prop.GW = self then
17:   RequestRelay(topic)        \>
18: end if
19: end for
20: end procedure

by distance function (Algorithm 9, lines 8 and 9). If the selected gateway, e.g., GW in Algorithm 9, is different from the current proposal, the node increases a counter inside the proposal for GW. This counter indicates the distance of the node to the GW, in terms of hop counts. If this distance exceeds a predefined threshold \(d\), the node ignores the proposal (Algorithm 9, line 10). A gateway node, therefore, is responsible for the nodes, which are a maximum of \(d\) hops away from it. Consequently, the number of gateways per cluster becomes proportional to the diameter of the cluster, and can be controlled by the distance threshold \(d\). That implies the worst case propagation delay inside a cluster is bounded to \(d\). Hence, the propagation delay in Vitis is \(O(\log^2 N + d)\). Nevertheless, \(d\) is a constant that does not depend on \(N\) and in all the practical scenarios, it can be set to a value less than \(\log^2 N\). Therefore, the overall propagation delay is bounded to \(O(\log^2 N)\). As our experiments show, in practice this value is much smaller than this upper bound.

When a node recognizes itself as gateway for topic \(t\) (Algorithm 9, line 20), it initiates the relay path construction by performing a lookup on \(hash(t)\). Since all the lookups end up at the rendezvous node (the lookup consistency is ensured by the ring), all the clusters of topic \(t\) get connected.

It is important to note that nodes do not need to reach consensus on gateways and multiple gateways can be selected for each cluster. This results in establishment of several relay paths from the same cluster and, therefore, more traffic overhead. However, it does not affect the correctness of the solution and is beneficial because: (i) the overlay becomes more robust, in particular to the failure of gateway nodes
or relay nodes along the path, and (ii) the propagation delay inside the cluster decreases, since the events are flooded simultaneously in different parts of the cluster.

Should a gateway node fail or disconnect from the cluster (e.g., due to a change of priorities that are enforced by the preference function), its immediate neighbors would detect the failure (after not receiving the heartbeat messages) and stop proposing it as a gateway. Therefore, in the proceeding rounds, those nodes select a different gateway.

**Event Dissemination**

Whenever a node publishes an event on a topic, it sends a notification to those neighbors in its routing table, which are interested in that topic, or act as a relay node for the topic. A node that receives a notification, pulls the event from the sender and forwards the notification to all its own interested neighbors. As a result, the notification propagates inside the cluster of the publisher node. When the notification is received by the gateway node, it is forwarded along the relay path. The notification goes up to the rendezvous node and again down the other existing relay paths, if any other cluster for that topic exists. It, then, reaches the gateway node(s) of those clusters, and will be flooded inside those clusters, accordingly. Figure 5.3 shows an example of how a notification is disseminated in the overlay. Node \( p \) publishes a new notification on topic \( t \), and sends it to all its neighbors, which are interested in \( t \). When this notification is received by the gateway node, \( g \), it is forwarded on the relay path towards the rendezvous node, i.e., node \( t \). When node \( t \) receives the notification, it sends it to the other existing relay path. Consequently, node \( m \) is informed and propagates the notification inside its own cluster. The event is pulled from the same path as the notification propagated along.

**Overlay Maintenance**

We use a mechanism similar to T-Man [16] and Scribe [64] for maintaining the routing tables and relay paths, respectively. Every time a node sends its profile to its neighbors, it increments the age of those neighbors (Algorithm 10). When it receives back a response from the neighbor, it marks that neighbor as fresh, by resetting its age to zero (Algorithm 11, line 3). After a predefined threshold, the stale entries are removed from the routing tables. This threshold determines the failure detection speed. The lower the threshold, the faster the failure detection is. However, if the threshold is too low, then the rate of false positives, due to the congestion in the network and varying link delays, increases. By increasing the threshold, the responsiveness of the failure detection can be traded off for more accuracy.

As we described earlier, the overlay is constructed by gossiping. Through gossiping, clusters are formed, gateway nodes are selected, and relay paths are established. The overlay maintenance is conducted in exactly the same way. When a node leaves
Figure 5.3: Node $p$ publishes a notification inside its own cluster. The notification is flooded inside the cluster. It is also forwarded to the relay node $t$ through the gateway $g$. The notification moves along the relay path up to the rendezvous node $r$, and then reaches the other existing clusters. Next, it is flooded inside those clusters.

Algorithm 10 Exchange Profile - Active

1: procedure ExchangeProfile
2:   profile ← UpdateProfile()
3:   for all neighbor in RT do
4:     if neighbor.age > THRESHOLD then $\triangleright$ remove the stale neighbors
5:       RT.remove(neighbor)
6:     else
7:       RT.neighbor.IncrementAge()
8:       Send [profile] to neighbor
9:   end if
10: end for
11: end procedure

Algorithm 11 Exchange Profile - Reactive

1: procedure RespondToExchangeProfile
2:  Recv profile from neighbor
3:   RT.update(neighbor, profile, 0) $\triangleright$ 0 indicates the age of this neighbor
4: end procedure

the system or modifies its subscriptions, the friend selection mechanism in the proceeding rounds captures this change and routing tables are updated accordingly. If the node is a gateway, then its direct neighbors in the corresponding cluster will notice the change and revise their proposals for selecting a new gateway. If the node is a relay node or rendezvous node, the proceeding lookups by their neighbors
on the relay path, will return a substitute node. Consequently, the overlay adapts to the changes in the network, while nodes constantly acquire fresh information through their neighbors.

5.4 Experiments

We implemented Vitis and two base-line solutions in Peersim [82], a simulator for modeling large scale peer-to-peer networks. The base-line solutions are:

- **RVR**: a structured RendezVous Routing solution that builds a multicast tree per topic, equivalent to that of Scribe [64] or Bayeux [65], with fixed node degree.
- **OPT**: an unstructured subscription aware solution that constructs an Overlay Per Topic, while minimizing node degrees by exploiting the subscription correlations, similar to SpiderCast [74].

To make the three systems comparable they use the same peer sampling service (Newscast [81]) and overlay construction protocol (T-Man [16]).

We evaluate Vitis against RVR and OPT with subscription patterns, generated from a synthetic model as well as real-world Twitter traces [54]. We investigate the impact of varying publication rates and routing table sizes on the performance of the systems. Moreover, the robustness of Vitis under churn is evaluated by utilizing traces from Skype [68].

In our simulations, we measure the following metrics:

- **Hit ratio**: The fraction of events, on all topics, that are received by the subscriber nodes;
- **Traffic overhead**: The proportion of relay (uninteresting) traffic that nodes experience;
- **Propagation delay**: The average number of hops events take to reach to all the subscriber nodes.

**Experimental settings**

We measure the performance of Vitis, RVR, and OPT with 10,000 nodes. Unless otherwise mentioned, $k$ is set to 3, the routing table size is set to 15, $d$ is set to 5, and different topics have the same rate of publication.

We generate three subscription patterns to model different levels of interest correlation. This data generation model was inspired by a work of Wong et al [83]. The subscription patterns are:

- **Random**: nodes select 50 out of 5000 topics uniformly at random;
5.4. EXPERIMENTS

![Figure 5.4: Measurements with varying number of friends](image)

- **Low correlation**: nodes group 5000 topics into 100 buckets and select 50 topics uniformly at random from 5 different buckets (10 topics from each bucket);

- **High correlation**: nodes group the 5000 topics into 100 buckets and select 50 topics uniformly at random from 2 buckets (25 topics from each bucket).

Note that, in all the above subscription patterns, the average topic popularity, i.e., the population of nodes subscribed to a topic, is uniform. Whereas, the distribution of interest correlation, captured by Equation 5.1, is different in the three patterns. Since RVR exhibits similar behavior with random and correlated subscriptions, we draw only a single line for it in the plots. Moreover, since SpiderCast is targeted for real-world scenarios with high subscription correlation, we investigate the performance of OPT only with Twitter subscriptions.
Friends Vs. sw-neighbors

In this experiment, we investigate the performance impact of varying the number of friends versus sw-neighbors. We bound the node degree to 15, that is, each node has a routing table of size 15, among which two links are dedicated for the predecessor and the successor of the node. That means, nodes have at least two sw-neighbors in all the experiments. The rest of the links can be selected, either as a friend or as sw-neighbor.

The results showed that both Vitis and RVR have 100% hit ratio in all settings. As we observe in Figure 5.4a, when more friends are selected, the traffic overhead in Vitis drops significantly. With correlated subscriptions, this traffic reduced by a factor of 88%. Even when the subscriptions are random, the traffic overhead in Vitis is less than one third compared to that of RVR. That shows Vitis is able to exploit even the slightest similarities between nodes subscriptions.

As it is shown in Figure 5.4b, nodes with correlated subscriptions experience a better delivery time as well. The propagation speed improves when more friend links are selected. This is due to the fact that selecting more friends results in a better clustering of nodes with similar subscriptions. Thus, instead of having many small clusters, the overlay moves towards having fewer, but bigger clusters. Since the events are very quickly disseminated inside clusters (by flooding), most of delay is caused by the inter-cluster routing. Therefore with fewer clusters, the event dissemination happens much faster. For random subscriptions, however, the overlay ends up having multiple small clusters per topic. Therefore, inter-cluster routing plays an important role for delivering the events to the subscriber node. Since replacing sw-neighbors with friend links degrades the navigability of system, the improved traffic overhead in this case, comes at the cost of higher propagation delay. However, as discussed in section 5.3, the propagation delay in our system is bounded to $O(\log^2 N)$.

Moreover, one might argue that although the average traffic overhead is reduced in Vitis, a high load is imposed upon gateway nodes, rendezvous nodes, or other relay nodes. Therefore, we show the traffic overhead distribution among the nodes in the overlay. Figure 5.5 shows that while the fraction of nodes with 10% overhead is increased, the fraction of nodes that have an overhead more than 20%, drops to less than one third in Vitis, compared to that of RVR. This shows that Vitis, not only reduces the average traffic overhead, but also improves the distribution of this traffic among the nodes.

In the rest of our experiments we set one predecessor, one successor, and one sw-neighbor for each node. The rest of the links are selected as friends.

Changing the routing table size

In this experiment we compare the performance of Vitis to RVR, while changing the routing table size from 15 to 35. As it is shown in Figure 5.6a, when nodes maintain bigger routing tables, the traffic overhead, as well as the propagation
delay, decreases in both systems, though, for different reasons. In RVR, this improvement is because the rendezvous routing performs better, i.e., in fewer number of hops, with more small-world links. Thus, more efficient spanning trees with less intermediary nodes are constructed. In Vitis, however, the number of sw-neighbors are fixed and the additional entries in the routing tables are used for adding friend links. Therefore, nodes are grouped together more efficiently and fewer relay paths per topic are required. This means inter-cluster routing constitutes a smaller part of the event dissemination. This explains why event delivery latency in Vitis with random subscriptions, outperforms the RVR system, when the routing table size exceeds 30 entries.
CHAPTER 5. VITIS

Changing the publication rate

So far we have assumed a uniform distribution of published events on each topic. However, the publication rate of topics does not have to be uniform. In fact, usually there are a few hot topics with a high rate of publications, while other topics have a low publication rate. In this experiment, we show how our solution adapts to different publication rates. We employ a power-law function, with a parameter $\alpha$, to define the distribution of events rate on different topics. We change $\alpha$ from 0.3 to 3, and evaluate the behavior of Vitis versus RVR. Note that the X-axis in Figure 5.7 is in the log scale. When $\alpha$ is close to 0.3, the distribution is similar to a uniform distribution as in the previous experiments. However, when $\alpha$ increases the distribution becomes more skewed. In the extreme case, when $\alpha$ is 3, almost all the events are published on a single topic.

When the publication rate for different topics becomes more skewed, Equation 5.1 gives a higher utility value to the nodes that are interested in the hot topics. Thus, such topics end up having fewer and better connected clusters. This effect is similar to when the correlation level is increased. That is why in Figure 5.7, the performance of the scenario with random subscriptions gets closer to that of the scenario with high correlation, when $\alpha$ is increased.

Note that, while hot topics are prioritized, topics with less events might experience higher traffic overhead and propagation delay. However, since hot topics constitute most of the published events, and they are propagated efficiently, an overall improvement is achieved.

Real world subscriptions

In this experiment, we evaluate Vitis with both RVR and OPT. We use a subscription pattern extracted from nearly 2.4 million Twitter users [54]. Each node in Twitter plays a dual role, that is, it can follow (subscribe to) other nodes, and it can be followed by others (as a topic). Thus, both topics and nodes refer to the users of the system. We analyzed the available data set and came up with the statistical results reported in Figure 5.9. The distribution of nodes in-degree and out-degree are modeled by a power-law distribution with an estimated parameter of 1.65 (Figure 5.8).

We took a sample of nearly 10000 nodes, by performing multiple breath first searches (BFS) [84]. Initially we randomly selected a number of nodes from the dataset. Then we added to this sample, all the subscriptions of these nodes, i.e., nodes being followed by the selected nodes. Next, we extracted all the relations (following or being followed) between these nodes. Finally, we removed subscriptions to the nodes outside the sample. In order to ensure that this approach preserves the properties of the complete log, we took several samples and the similarity of in-degree and out-degree distribution of the samples and that of the full log was confirmed.

Unlike our previous configurations, in these experiments the number of sub-
5.4. EXPERIMENTS

![Distribution of in-degree and out-degree in Twitter](image)

**Figure 5.8:** Distribution of in-degree and out-degree in Twitter

![Summary of statistical analysis of available Twitter data set](image)

**Figure 5.9:** Summary of statistical analysis of available Twitter data set

subscriptions per node is not the same. We changed the routing table size from 15 to 35 and investigated the impact on the hit ratio, traffic overhead and propagation delay. Moreover, we measured the hit ratio for a node, 10 seconds after the node joins the system. That means a node is expected to receive the subscribed-to events, which are published 10 seconds after its joining time.

Figure 5.10a shows that the hit ratio in Vitis and RVR is 100%, while OPT with a bounded node degree can not achieve a full hit ratio. Even when the node degree is 35, OPT can only hit 80% of the subscribers, on average. In order to reach a 100% hit ratio, OPT needs to be free of any bound on the node degree. We performed another experiment to investigate the performance of OPT with unbounded node degree, and plotted the node degree distribution in Figure 5.11. As can be seen in this figure, more than two third of the nodes have a degree higher than 15. Also, 0.3% of nodes have a degree higher than 200 (Maximum observed degree is 708), which is not shown in the figure. This implies OPT-like solutions, that only rely on exploiting subscription correlations, can not scale in real world scenarios.

In contrast, OPT outperforms Vitis and RVR with respect to traffic overhead. Since OPT constructs a separate overlay per topic, the events are only disseminated among the subscribers and there is no traffic overhead at all. Figure 5.10b shows the traffic overhead of the three systems. As it is shown, Vitis and RVR has a higher level of overhead compared to Figure 5.6a, which is due to the increased number of subscriptions (on average 80 subscriptions per node). Also, the number of topics in this experiments is doubled, since there are as many topics as the number of nodes. Therefore, the average population of nodes that are interested in a topic is less than the previous experiments. However, even with only 15 links per node,
Vitis has 30% less traffic overhead compared to RVR. With 35 links per node, the traffic overhead in Vitis decreases to 43%, which is 40% better than RVR.

The propagation delay in all three systems exhibits a similar trend when the routing table size increases (Figure 5.10c), while Vitis is more than 1.5 times faster than RVR and 1.7 times faster than OPT. Note that due to the navigable structure, the delay in Vitis and RVR is bounded. However, a topic overlay in OPT might be any arbitrary graph and therefore there is no upper bound on the propagation delay.

Vitis under churn

In this experiment, we use a scenario with churn, i.e., a scenario in which nodes can join or leave at any time. We use a real world trace [68], which monitors a set of 4000 nodes participating in the Skype superpeer network for one month beginning September 12, 2005. The routing table size is bounded to 15, and a uniform publication rate for the topics is considered. Like the previous experiment, the hit ratio for a node is calculated 10 seconds after the node joins the system. We
5.5 Conclusion

We presented Vitis, a topic-based publish/subscribe system, which scales with the number of nodes as well as the number of topics in the overlay.

The main contribution of this paper is a novel hybrid publish/subscribe overlay that exploits two ostensibly opposite mechanisms: unstructured clustering of similar peers and structured rendezvous routing. We employ a gossiping technique to compare Vitis with RVR and observe that, due to the underlying gossip mechanism, both solutions react nicely to the churn and adapt to the changes of network.

As Figure 5.12a shows, although both systems can tolerate moderate churn, under flash crowds, i.e., a large number of nodes join at nearly the same time, the hit ratio in RVR goes down to 87%. That is because the stabilization time takes longer, and while the structure is not converged to a connected subgraph per topic, nodes may miss some events. This effect is also observable in our system. However, the worst case hit ratio is about 99%. This is because as soon as a node finds a group-mate for a topic, it can receive the corresponding events on that topic, without the need for establishing a relay path independently.

We also observe in Figure 5.12b that the traffic overhead in both systems does not change much over time. However, under flash crowds, the traffic overhead in RVR drops sharply. This is not an advantage though, because the relay paths are not established properly and nodes are missing their desired events (that is why the hit ratio drops as well). In contrast, the traffic overhead in Vitis slightly increases under flash crowds, because nodes inside the groups are not yet informed about their group-mates and therefore several gateway nodes start to build up the relay paths towards the rendezvous point. After a while, however, when the churn is moderate, the number of gateways and, consequently, the traffic overhead decrease. Likewise, Figure 5.12c shows that the propagation delay does not change in moderate churn. However, the increased level of delay after the flash crowd is due to the bigger size of the network.
embed a navigable small-world network, which efficiently establishes connectivity among clusters of nodes that exhibit similar subscriptions. We also give a theoretical bound on the worst case delay.

We showed that Vitis fills in the gap in the range of solutions, by simultaneously achieving both bounded node degree and low traffic overhead. We evaluated Vitis, in simulations, by comparing its performance against two base-line solutions, which represent two main groups of the related work: a structured overlay that uses rendezvous routing (RVR), and a solution that takes advantage of subscription

Figure 5.12: Measurements with Skype trace for churn in the network
similarities to constructs an overlay per topic (OPT).

We used synthetic data as well as real-world traces from Twitter to model users subscriptions. We also used traces from Skype to show that Vitis is robust in the presence of churn. We showed that although exploiting subscription correlations results in great advantages, solutions such as OPT, which solely rely on such correlations, can not scale when the number of node subscriptions increases. Consequently, in real world scenarios, such solutions cannot guarantee that the subscribers receive their intended data, unless the node degrees are unbounded. In contrast, Vitis and RVR always reach a perfect hit ratio. This, however comes at the cost of some traffic overhead. We showed that, compared to the rendezvous routing solution, Vitis reduces the traffic overhead to less than 75% with synthetic data and 40% for real-world traces, while it speeds up the event dissemination in the overlay. Moreover, Vitis adapts to biased rates of events that are published on different topics, and builds more efficient groups for hot topics, thus, improving the overall performance of the event dissemination.
Chapter 6

Locality-awareness in a Peer-to-Peer Publish/Subscribe Network

Fatemeh Rahimian, Thinh Le Nguyen Huu, Sarunas Girdzijauskas

In the 12th International Federated Conference on Distributed Computing Techniques (DiscoTec, DAIS), June 2012, Stockholm, Sweden.
Locality-awareness in a Peer-to-Peer Publish/Subscribe Network

Fatemeh Rahimian†‡, Thinh Le Nguyen Huu† and Sarunas Girdzijauskas†

†Swedish Institute of Computer Science (SICS)
‡KTH - Royal Institute of Technology
{fatemeh, sarunas, thinh}@sics.se

Abstract

Peer-to-peer publish/subscribe systems are promising solutions to provide distributed content distribution services at Internet-scale with low cost. One of the potential problems with peer-to-peer overlays, however, is the inefficient traffic and large delays, due to the mismatch between the physical network and the overlay topology. This paper introduces a locality-aware extension to a peer-to-peer publish/subscribe system, named Vitis. The ultimate purpose is to avoid communications over long-distance links, instead, nodes send data over short-distance and low-cost links, when possible, while maintaining an acceptable quality of service. We show, through simulations, that the average data delivery time is up to 40% improved. The cost to pay is at most 10% more relaying in the peer-to-peer overlay.

6.1 Introduction

Distributed applications and services are getting significantly popular over the Internet. Social networks, news syndications, and multi-player on-line games are just a few such services one may use on a daily basis. One of the dominant communication protocols, used in many of these applications, is the publish/subscribe protocol. Work on the distributed publish/subscribe models, therefore, has attracted many researches in the field and several interesting solutions have been introduced [31, 62, 70-72, 74, 79, 85]. In particular, peer-to-peer publish/subscribe systems, such as [31, 62, 74, 79, 85], exploit the increasingly available resources at the edges of the network, in order to provide a scalable and reliable, yet cheap service to millions of users over the Internet. Although a few of these solutions are proven to be effective, a problem that exists in many peer-to-peer systems in general, is the mismatch between the overlay topology and the topology of the underlying network. Neglecting this mismatch in data-intensive applications could render these systems very inefficient. Since publish/subscribe systems are all about content distribution, and this content could be potentially massive, it is important to design proximity-aware publish/subscribe overlays.

In this paper, we propose a solution for incorporating locality-awareness into an existing topic-based publish/subscribe system, Vitis [31]. Currently in Vitis,
nodes build up clusters by taking advantage of the subscription correlations. More precisely, a node selects its neighbors, i.e., the nodes it directly connect to, based on the similarity of subscriptions. If two nodes have more topics in common, they have a better chance to become neighbors. Vitis nodes, however, are oblivious to node geographical positions, which may result in large traffic in the physical network.

The core of our solution is to get nodes gain more information about the location of the other nodes, thus, enabling them make a better decision on where to forward the data. As a result, most of the traffic is likely to be confined among nodes that are co-located in the same ISP or the same geographical area, and there is less communication between nodes lying on different continents or countries. Average delay to get some data delivered is also reduced, since a large portion of data forwarding is on the short links.

Through simulation and by using synthetically generated subscription, as well as real-world subscriptions from Twitter, we show that the average delivery time of the published events is improved by up to 40%. This improvement, however, comes at the cost of (at most) 10% more relay traffic overhead in the overlay. Since the total relaying traffic is only 4% of the whole overlay traffic, we conclude that our solution pays off, as it brings along up to 40% improved delivery time, which translates into more efficient communications in the physical network.

In the next section we briefly go through some of the related work for locality awareness in peer-to-peer networks. In Section 6.3, we first explain how Vitis works currently, and then elaborate on our technique to make the overlay locality-aware. In Section 6.4 we report the results of our experiments and in Section 6.5 we conclude the paper.

6.2 Related Work

In [86], X. Y. Zhang and Q. Zhang suggested mOverlay, an efficient overlay network, with locality of network hosts, that reduces the communication delay and cost. In order to gain information about the position of nodes, mOverlay utilizes a set of dynamic landmarks to measure the distance of nodes and group the close ones together. Having the estimated distances, nodes are assigned to the closest group they could find, thus, locality-awareness in the overlay is improved. Although mOverlay has shown improvements in gaining locality-awareness, it does have a few limitations and weaknesses. The group that nodes fall into, is dependent on the initial nodes that are introduced to it. Since the group choosing process terminates after a number of rounds, it may leave the new node in a group far from the optimal one. Furthermore, mOverlay does not mention anything about upper bound of a group, thus, some groups may get over crowded, while others are quite empty. Skewed distribution of nodes makes the cost of topology maintenance and cache exchanging within a group high. Lastly, the solution uses timestamps for selecting new group leader, and for this, it assumes nodes have synchronized clocks, which is not the case in reality.
Yuhao Liu et al. in [87] presented another solution for this problem. The solution, which is called *location-aware topology matching (LTM)*, continuously reconstructs the overlay by disconnecting slow connections and choosing physically closer nodes as logical neighbors. Similarly to previous solutions, nodes in LTM detect their distance to others by measuring roundtrip message delay. Based on this information, LTM nodes can detect and cut most of the expensive and redundant links, thus makes the topology more efficient. The problem with LTM is when nodes do the optimization, a node A may remove the link between itself and another node B without knowing that this link is essential to node B. This could leave node B disconnected for a while until the next round when node B tries to reestablish the connection to A. The process of removing and creating link can go on for a long time until there are significant changes in connections that result in a different topology. Thus, it creates an overhead in the cost to maintain the overlay and the overlay changes all the time without making the topology better. Also, LTM does not help the new nodes to initially acquire an appropriate position in the overlay. Optimization process happens mainly on constructed overlay, in which nodes are randomly connected. Thus, it may take a long time before the overlay stabilizes. Furthermore, LTM only works on a small scale and does not fully optimize the whole peer-to-peer network, resulting in a network that is partially optimized here and there but not on the whole.

Using an approach similar to mOverlay [86], the authors of [88] and [89] proposed a solution, named *distributed binning* to arrange nodes into areas that will exploit the geographical information to gain locality-awareness. The idea behind this is that nodes are arranged into bins, with unique identifiers. Nodes within the same bin are connected using, so called, short links, and communications over short links are rather cheap and fast while links connecting bins together (long links) are more expensive. Ideally, there should be only one single link between two bins, however redundancy could be used to make the system more robust against failure. As before, network latency is calculated by counting the average time of round communication, for instance the average of 10 pings. Unlike [86], distributed binning uses a set of static landmarks to determine the relative position of nodes. Any static systems such as DNS root servers or a set of well-known, widely distributed servers could be used for this purpose. On the scale of the Internet, it is estimated that 8 to 12 landmarks is enough for the system. This solution experiences the similar trade-offs to those that use dynamic landmarks. First of all, since the landmarks are static and the latency level division is predefined, bins are rather static and not very flexible. This leads to the situation that some bins are over-crowded than the others and maintaining the overlay within those bins becomes too expensive. It is also unclear that if the layer vectors of two nodes do not share any common landmarks, how to estimate the relative position of the two bins. Those two nodes could end up in different bins even though the two bins are really close to each other. Finally, congestions and temporary network delay could result in inaccurate bin determination thus, leaving the a new node in a completely wrong bin and it
never gets to the correct one. This renders the topology inaccurate and inefficient.

Another approach is to embed geographical position into node identifiers as proposed in [90–92]. In those solutions, a node ID can have hierarchical prefixes representing regions, sub-regions, countries, ISPs and actual node ID. Nodes in the same area will share some similar bits. This is rather a static solution and it requires a policy to distribute node IDs among regions based on the population of nodes so that there are enough node ids for a crowded area, while not allocating too many for sparse ones.

6.3 Locality-aware Publish/Subscribe

Previous studies to improve locality-awareness have many flaws and limitations. Some constructs an overlay that is too complicated and too costly to maintain. Some are inefficient and only improve locality-awareness of some areas in the overlay rather than a complete solution on a large scale. In this paper, locality-awareness improvement is studied based on Vitis [31], a gossip-based [16] hybrid overlay for Internet-scale publish/subscribe developed by Rahimian et al.

Vitis - a topic-based publish/subscribe system

Vitis [31] is a topic-based peer-to-peer publish/subscribe solution that requires only a bounded node degree, but generates very low traffic overhead, compares to its counterparts. It uses a novel technique for overlay construction, in which nodes exploit the subscription similarities and select as neighbors, nodes with whom they share the most topics. It denotes a cluster for a topic as a maximal connected subgraph of the overlay, which includes a set of nodes that are all interested in that topic.

Due to the bounded node degree requirement, there is no guarantee that all the nodes, which are interested in a topic, connect together. In fact, any number of clusters for the same topic can emerge in different parts of the overlay. Although nodes inside a cluster are reachable from one another, in order to make sure a published event for a topic is delivered to all the subscribers, all the clusters of that topic must be also linked together. The path that connects different clusters of the same topic is called relay path. Such a path includes nodes that are not interested in the topic themselves. We refer to these nodes as relay nodes, hereafter. The challenge is to decrease the number of required relay nodes, while making sure that all the clusters associated with a topic (and therefore, all the nodes interested in that topic) are linked together. To enable relaying between the clusters, vitis uses rendezvous routing [66] on top of an unstructured overlay. The node that has the closest identifier to a topic identifier is designated as the rendezvous node for that topic.

Moreover, Vitis nodes utilize a novel algorithm to select a number of representative nodes, as gateways, in each cluster. The number of gateways for a cluster is proportional to the diameter of the subgraph that represents the cluster. Gateway
nodes are responsible for employing the navigable small-world overlay to connect to other clusters for the same topic. They perform a greedy lookup for the topic identifier, and all meet at the same node, i.e., rendezvous node. The structure of Vitis is depicted in Figure 6.1.

Whenever a node publishes an event on a topic, it sends a notification to those neighbors in its routing table, which are interested in that topic, or act as a relay node for the topic. A node that receives a notification, pulls the event from the sender and forwards the notification to all its own interested neighbors. As a result, the notification propagates inside the cluster of the publisher node. When the notification is received by the gateway node, it is forwarded along the relay path. The notification goes up to the rendezvous node and again down the other existing relay paths, if any other cluster for that topic exists. It, then, reaches the gateway node(s) of those clusters, and will be flooded inside those clusters, accordingly.

To enable an efficient clustering, Vitis nodes use a utility function to rank any arbitrary nodes that they come across. The ranking is based on the similarity of interests between the two nodes, which in turn, defines the benefits of establish a connection between the two nodes. Every node ranks the nodes it learns about, and selects the highest ranked nodes as its neighbors in the overlay. By doing so, a node ensures its neighbors will have the most similar topics to itself and an interesting event is more likely to be interesting for its neighbors too, thus keeping relay traffic overhead very low. Every node also selects a few small-world connections [67] for the sake of correctness and efficiency of routing in the overlay.

**Locality-awareness improvement in Vitis**

Vitis can further be improved to deliver a better performance in terms of the actual network traffic. Currently, Vitis puts the nodes with similar interests into clusters
and most of the data dissemination happens inside these clusters. Although this optimizes the amount of traffic in the overlay network, it is not optimal for the actual network traffic, as nodes far apart in the physical network may be positioned in the same cluster and exchange huge amount of data, in which they have a common interest. Communication between nodes, thus, becomes costly, high-delayed and bandwidth-limited. On the other hand, connecting nodes to the nodes with close-by geographical coordinates, does not necessarily result in a lower communication cost at the end, because the traffic is not anymore confined to efficient clusters, and many uninterested nodes may have to relay data to ensure that it reaches all the corresponding subscribers. Hence, we will have more relay traffic and a longer delivery time.

Hence, to improve the performance of Vitis, it is necessary to maintain the clusters of nodes with similar interests, while reducing the geographical divergence inside these clusters. Putting differently, we would like to enhance locality-awareness, while keeping the relay overhead trade-off at an acceptable level. As mentioned before, a number of connections in Vitis are chosen based on a utility value, and the utility value is calculated purely based on the similarity in node subscriptions. A node has no clues about where the other nodes are, thus its view does not reflect its actual physical connections. In order to make the overlay more similar to the physical world, nodes need to also consider the notion of distance when selecting their connections. Note that, small-world connections are selected just as before and remain untouched by this improvement.

We introduce a utility function that could dramatically change the behavior of the overlay and help enriching nodes with proximity knowledge. The new utility function is shown in 6.1.

\[
\text{utility}(i, j) = \left( \frac{1}{d(i, j)} \right)^n \cdot \frac{\sum_{t \in \text{subs}(i)} \sum_{\text{subs}(j) \cap \text{subs}(i)} \text{rate}(t)}{\sum_{t \in \text{subs}(i) \cup \text{subs}(j)} \text{rate}(t)}
\]

(6.1)

where \(i\) and \(j\) are two nodes in the system and \(\text{sub}(i)\) and \(\text{sub}(j)\) are the sets of subscriptions of the two nodes, respectively. Note, a subscription is a set of topics that a node is interested in. In reality, data/event about some topics are published more frequently than the other topics. The term \(\text{rate}(t)\) indicates the publication rate of events on topic \(t\). If a topic is hot and has many publications, the overlay builds more efficient clusters for it. Without losing the generality, we assume the publication rate of all topics are equal. The term \(d(i, j)\) denotes the distance between two nodes \(i\) and \(j\). This distance is measured as the round-trip latency. As a result, nodes with a long communication delay, which are very likely in different ISPs or distantly located, are less preferable. Now, the question is how to tune the effect of locality versus subscription similarity. In other words, we can have different weights (denoted as \(n\) in the utility function) for distance impact.
6.3. LOCALITY-AWARE PUBLISH/SUBSCRIBE

(a) Connections in the overlay network are oblivious to locality. Clusters are span across different ISPs, thus, many overlay links cross the boundaries of these ISPs.

(b) The overlay topology is locality-aware. Clusters are confined to the ISPs and few links cross the ISPs in order to connect the clusters together.

Figure 6.2: The impact of locality-awareness on the overlay topology. Nodes, colored in black, are all interested in the same topic. ISPs are shown in light gray boxes, and the red (dark gray in the printed version) regions represent the clusters in the overlay network.

relative to common interests impact, and we do not know yet what is the optimal configuration. In Section 6.4 we carried a set of simulations to select an appropriate value for $n$.

Nevertheless, the utility value is not only reflecting how similar the two nodes are, but also how close they are in the physical network. Link selection, thus, becomes more locality-aware and nodes make better decision in term of limiting the cost of communication. The closer two nodes node are, the better chance they have to establish a connection, if they have some common interests. On contrary, although two nodes may have many common interests, if they are so far away, there is very little chance that they get directly connected and stay in the same cluster.

Having a closer look at the topology of the overlay, we observe that, compared to the original Vitis, the clusters are now bigger in numbers, but smaller in size. Figure 6.2a shows a part of an original Vitis overlay, which does not take locality into account. Clusters are span across different ISPs, thus, many overlay links cross the boundaries of these ISPs. In the locality-aware version, depicted in figure 6.2b, however, overlay connections are not anymore oblivious to the geographical distance of the nodes, thus, clusters contain nodes that are close-by in both physical space and subscription space. Therefore, data dissemination mostly happens inside the clusters and very few links connect the clusters that reside in different ISPs. This not only generates far less traffic across ISPs, but also improves the delivery latency. However, since the number of clusters is increased, in order to connect all the clusters of a topic together, more relay nodes (gray nodes in figure 6.2b) might be involved. Putting differently, the price of improvements in the delivery time has to be paid by the increased overhead in relay traffic. Nevertheless, the traffic in the
underlying network is forwarded more efficiently. To better understand this trade-off, we run a series of experiments, which we will describe in details in Section 6.4.

6.4 Evaluations

We verify the correctness of our solution by simulating it on Kompics [93], a message-passing component model for building distributed systems. Simulations are done with several configurations and different input data. The configuration parameters are tweaked to find out the optimal settings that delivers the most improvement as well as the best trade-offs.

Experimental setup.

Simulations are ran with a peer-to-peer network that consists of up to 4000 nodes. To model node subscriptions, the following patterns are considered: (i) random, (ii) low-correlated, (iii) high-correlated, and (iv) Twitter subscription data. For the first three patterns, a pool of 1000 topics is generated randomly and nodes pick the same amount of topics as their interests. In random pattern, each node selects 50 topics randomly from the pool, thus, nodes subscriptions are quite diverging and they have almost no correlation. However, in low-correlated pattern, the topic pool is divided into 20 buckets, each node then picks five random buckets and selects ten topics from each bucket. For high-correlated pattern, the settings are two buckets and 25 topics from each. Differently from random patterns, nodes have more probability to share common topics and the subscriptions are more correlated. In the other words, two arbitrary nodes may share many topics since they are picked from the same bucket.

The simulation scenario features nodes joining with a uniform distribution in a deterministic way, inter-arrival time is 100 milliseconds. Even though there are many randomizations involved, the scenario can completely be reproduced again to guarantee identical set of inputs. After joining, node are given a period of time, warm-up period, to make connections, exchange views, set up relay paths as well as to gather into clusters. When the warm-up period is over, the topology is in a stable state and then 1000 events are published. Publishing is done by random nodes selecting a random topic in their interests and generating an event on that topic. Finally, when all events are published, there is a cool-down period until all events get delivered. Statistics of the run are gathered and logged for further analyzing and then the simulation terminates. Each experiment is run with and without locality-awareness and performances are put on the scale for comparison. We measure (i) the average delivery time of the published events and (ii) the average percentage of relay traffic load on each node.
Estimating the distance.

There are many ways to model distance between nodes. In our implementation, distances between nodes are modeled based on real-world measurements using a technique, called King [94]. Kompics provides us with a full latency matrix driven from King measurements. This latency matrix provides a node with the estimated distance to each and every of its neighbors. King works by approximating the latency between two end points by measuring the latency between nearby authoritative DNS name servers, using carefully constructed recursive queries. Compared to its counterparts, King is more accurate as it is based on direct, on-line measurements rather than off-line extrapolations. The King nodes are more geographically diverse than PlanetLab nodes, and the median round trip time of the King data set is 159ms [94]. Nevertheless, how the latency is measured is orthogonal to our work, and the resulting estimation, whatsoever, will act as an input to our algorithm.

Constructing the optimal utility function

The first set of experiments is to determine the optimal weight for the distance metric in our utility function. Power factor $n$ in formula 4.2 is tweaked with different values sampling as 1, 2, 5 and 10. The experiments are run with 500 nodes that subscribe to 50 topics randomly. In figure 6.3a delivery time improvements at different power factor are recorded and plotted on the graph. Trade-offs of the system are also plotted up in figure 6.3b Using power factor of 1 gives the best improvement in delivery time, roughly 37%. When power factor gets larger, distance seems to have less effect on the improvement. This is because a typical round-trip delay is around $100\text{ms}$, and when taking the reverse, it results in a rather small value. When the power factor goes up, the effect of round-trip delay (distance) gets small rapidly and the overlay is moving towards the case with no locality-awareness. Figure 6.3b shows a similar trend with traffic trade-offs. Power factor of 1 generates
10% more traffic overhead, the highest in all, while overhead drops down to only 2% at power factor 10. Choosing the power factor is then just a matter of determining whether the improvement is good enough to go over the trade-off. Because 10% of relay traffic is rather small and considering the significant improvement in delivery latency, we use the power factor 1 for our experiments, hereafter.

**Evaluation of different subscription patterns**

In this experiment, the network size is increased from 500 nodes to 4000 nodes, to study how the system behaves when the network gets more populated. We also observe the impact of several subscription patterns on the performance of the system.

Firstly, the impact of size and subscription pattern on the delivery latency with and without locality is shown in figures 6.4. The general trend for delivery latency...
latency, i.e., figure 6.4, indicates that the overlay routing is efficient, as it grows logarithmically with the network size. Without locality-awareness, random pattern has the largest delivery time, followed by the low-correlated pattern, and the high-correlated pattern has events delivered the fastest. Differences between the patterns at each point are roughly from $50\ ms$ to $100\ ms$, which is due to the more effective clustering in the presence of subscription correlation. When locality-awareness is added, the delivery latency in all subscription patterns reduces significantly. For example, latency in a network of 4000 nodes with random subscriptions drops from nearly $400\ ms$ to below $250\ ms$. Note, more correlated subscriptions still get a better performance, which means that the impact of clustering is preserved. As it is shown, locality-awareness improves the delivery latency between 25% and 40%, and the improvement increases with the network size.

We also measured the performance in terms of the relay traffic. As shown in figure 6.5, the generated relay traffic is more affected by the subscription pattern and the network size, whereas locality-awareness has a very little impact. Nevertheless, the relay traffic is increased by 10% on average. Note that, the relay traffic is decreased in larger networks, which is promising for scalability. It is also important to note, the increased relay traffic is in the overlay network, and not the physical network. In fact, the reduced delivery latency suggests that the routing in the physical network is performed more efficiently.

**Evaluation on real world data**

Previous experiments have proven that the proposed solution works efficiently on synthetically generated subscription patterns. However, in real world scenarios, subscription patterns are usually more complex: nodes subscribe to different number of topics and topics usually have a skewed popularity [63]. To evaluate the performance of our system with a real-world dataset, we use the Twitter subscription set [54, 63]. In Twitter any user can follow other people in order to receive their tweets (published events). Any user can also be followed by other users. That is, a user plays a dual role: a node and a topic. Twitter subscriptions have various sizes with a complex Poisson distribution with a very heavy tail. As there exist 2,484,449 records in the available data set and the simulation infrastructure cannot handle this many, we had to take a sample from this huge dataset. The sample, however, must represent the whole data set, i.e., it has to preserve the correlation between user subscriptions and maintain a comparable statistics. To make sure that we get an unbiased sample, we borrowed ideas from [84], and in a few steps collected the sampled nodes. Initially, we collected a number of Twitter nodes randomly from the whole dataset. We call this initial set the candidate set. Scanning through the whole dataset, we appended the set with all the nodes that are followed by the candidates, i.e., the subscription of the initial candidates were appended. We repeat the scanning and appending process until we collect the required number of sampled nodes. We then refine the set by removing all the nodes that either do not follow anyone in the current sample set or those that are followed by no one.
To make it even more realistic, one of the experiments is ran with churns. Churn is created by letting half of the nodes join and publish half of the events; the system then waits for some time before the rest of nodes join and the other half of events are published. The experiment is to verify if the system is robust against churns and if it is able to maintain full hit ratio. Unlike the previous simulations, this time, network size is fixed at 1000 nodes, while the number of events varies from 250 to 1000 events, with the step of 250.

Figures 6.6 and 6.7 show the performance of the system with Twitter subscription data. As expected, the performance is insensitive to the number of events. Delivery latency improvement, shown in figures 6.6, is very good both with or without churns (35% improvement on average). It is also interesting to see that when there is churn, delivery time is shorter than without churns. This is because nodes join in two batches and when the first batch joins, the network size is
smaller (half size), and therefore, the first set of events are delivered much faster. Figures 6.7 shows the relay traffic overhead, with and without locality-awareness. Relay traffic is less than 12% in all the cases and there is no big difference between data points.

6.5 Conclusions

Optimizing and reducing the costs in publish/subscribe systems is an interesting topic among many other challenging problems. This paper solidly focuses on making a publish/subscribe system, Vitis, ISP-friendly, in order to save network bandwidth and resources. Communications and event propagation favors short and cheap links rather than those long-delay, expensive connections. This is not only cost-wise but is also an improvement in system performance, avoiding bottle necks and utilizing local resources more efficiently.

The paper proposes a solution to embed locality information into the overlay topology, that is, to make the connections between nodes to better reflect the physical network connections. It introduces a notion of distance in the neighbor selection mechanism of Vitis and studies how we can exploit locality-awareness in-line with the subscription correlations. In other words, we investigate the trade-off between delivery latency and relay traffic with various subscription patterns, derived from real-world subscription, as well as, synthetically generated.

The simulation results showed that the system with locality-awareness outperformed the original system, in terms of delivery time, regardless of the subscription pattern. Simulations with synthetic patterns, i.e., random, low-correlated and high-correlated resulted in improvements ranging from 25% to 40%. With real-world Internet-scale data from Twitter application, the system still performs nicely with averagely 35% improvement. However, faster delivery has to be paid off by a slight increase in the amount of uninteresting traffic, i.e., relay traffic, in the overlay network. However, relay traffic over-head is negligible. Even when the subscriptions are more complex, such as the scenario with Twitter data and the churn, increments of relay traffic are kept very low, at 2% to 4%. This traffic, however, is in the application layer, while the reduced delivery latency shows that the actual routing in the physical network is done more efficiently.
Chapter 7

Subscription Awareness Meets Rendezvous Routing

Fatemeh Rahimian, Amir H. Payberah, Sarunas Girdzijauskas, and Seif Haridi

In the 4th IARIA International Conference on Advances in P2P Systems (AP2PS), Barcelona, Spain, September 2012.
Subscription Awareness Meets Rendezvous Routing

Fatemeh Rahimian†‡, Amir H. Payberah†
Sarunas Girdzijauskas‡ and Seif Haridi†

†Swedish Institute of Computer Science (SICS)
‡KTH - Royal Institute of Technology
{amir, fatemeh, sarunas, seif}@sics.se

Abstract

Publish/subscribe communication model has become an indispensable part of the Web 2.0 applications, such as social networks and news syndication. Although there exist a few systems that provide a genuinely scalable service for topic-based publish/subscribe model, the content-based solutions are still suffering from restricted subscription schemes, heavy and unbalanced load on the participating nodes, or excessively high matching complexity. We address these problems by constructing a distributed content-based publish/subscribe system by using only those components that are proven to be scalable and can withstand the workloads of massive sizes. Our publish/subscribe solution, Vinifera, requires only a bounded node degree and as we show, through simulations, it scales well to large network sizes and remains efficient under various subscription patterns and loads.

7.1 Introduction

The amount of data in the digital world that surrounds us is increasing very rapidly, thus, finding the relevant pieces of information becomes even more challenging. Publish/subscribe systems, which are now pivotal to many Web 2.0 applications, especially On-line Social Networks (OSNs) like Twitter, Facebook and Google+, leverage this problem by providing users with only the information they are actually interested in, e.g. a friend’s status update, sport news, or a music playlist. As a result, when some new data is generated, the interested subscribers are notified. All these examples, which classify data into coarse-grained predefined categories or topics, are known as topic-based subscriptions. On the other hand, subscriptions that define a more fine-grained filter over the content of the generated data, are called content-based. These subscriptions usually consist of continuous ranges of values over several attributes that describe the content. For example, a user may be interested to get notified if the local temperature is below zero between 6am and 6pm.

The traditional model to provide a publish/subscribe service uses a central server or broker to maintain node subscriptions \cite{95} [96] [97]. The published data is also sent to this central point and is matched against the existing subscriptions. Since
subscription maintenance and matching are done centrally in this approach, the server could become a bottleneck as the number of users and subscriptions grow. Consequently, researchers have studied distributed publish/subscribe systems, in particular peer-to-peer (P2P) solutions, as an alternative design paradigm to the centralized model. Currently, a wide range of solutions are proposed for topic-based publish/subscribe over P2P overlays [74] [31] [79] [62]. The topic-based solutions, however, can not be readily reused for the content-based model, due to the conceptual differences, i.e., discrete independent topics versus multiple continuous ranges over various attributes. Hence, a number of solutions have been proposed particularly for P2P content-based publish/subscribe [98] [99] [100] [101], spanning from the designs based on unstructured gossip driven overlays to highly structured overlays with rigid event dissemination structures.

In particular, at one end of the design space we find a family of solutions that are subscription aware [98] [99]. These solutions, partition the data space into subspaces that include each and every subscriber that is interested in that subspace. Published data is routed to the subspace that it belongs to, and is delivered to the subscribers in that subspace. As we describe in Section 7.2, these systems perform well under simple workloads, but fail to deliver an efficient service to massive number of users with multi-dimensional subscriptions, mainly because they require unbounded number of connections per node. Moreover, despite having to maintain potentially numerous connections, these solutions can not provide an upper bound guarantee for average delivery path length.

At the other end of the design space lie DHT-based solutions, such as [100] [102] [101], that exploit a technique, known as rendezvous routing [66]. To enable rendezvous routing all the nodes and attributes are embedded in an identifier space, by taking a random identifier. Also, a distance function is introduced to make a greedy routing possible. The node with the closest, but higher, identifier to the attribute identifier is selected as the responsible node, i.e., the rendezvous node, for that attribute. Every subscriber node that performs a greedy routing (lookup) for an attribute identifier, therefore, ends up at the rendezvous node for that attribute. Next, the reverse routing path, i.e., from the rendezvous node to the subscriber node, is used for data dissemination. Consequently, the dissemination structure consists of a single tree per attribute, thus, often consisting of a handful of dissemination trees for the whole node population. Note, this is different from the multiple rendezvous based trees for topic-based subscription model (e.g., as in Scribe [85]), because, as opposed to the topic-based model, where a subscriber to a topic wants to receive all the events relevant to that topic, in the content-based model nodes that join an attribute tree, are often subscribed to only a subset of the possible values for that attribute. Hence, while this approach does not require an unbounded node degree, the constructed dissemination trees, which blindly deliver all the events to every node on the tree, are very inefficient.

In fact, the aforementioned state-of-the-art solutions are forced to choose a trade-off between scalability (bounded node degree and efficient routing) and overhead (both in volume and distribution), thus, fail to provide a genuinely distributed pub-
lish/subscribe service for Internet-scale applications. Such state of the design space inspired us to work on a solution which would not require unsuitable trade-offs and could retain all the desirable properties of a scalable system under any scenarios. As a result, we propose Vinifera, which to the best of our knowledge, is the first P2P content-based system that simultaneously fulfills all the fundamental requirements of a scalable distributed service.

Vinifera is empowered by a gossip-based topology management service and clusters the nodes with similar subscriptions. These clusters are later exploited to create efficient data dissemination structures. The same gossiping process, also, embeds a navigable small-world structure into the overlay after each node is assigned an identifier, selected from a globally known identifier space. This enables a distributed greedy distance minimizing routing algorithm to find short paths between any two nodes, which in turn, allows us to utilize the aforementioned rendezvous routing technique [66]. However, in contrast to other content-based publish/subscribe systems that construct a single delivery tree per attribute, thus, suffer from an unbalanced load and large traffic overhead, Vinifera constructs a forest per attribute, where the roots of the trees in the forest are the rendezvous nodes for the attribute values, thus, the load is distributed over all the participating nodes. These trees are dynamically constructed based on the user subscriptions.

Vinifera forest is constructed by utilizing an order preserving hash function [103], that maps each and every attribute domain to the node identifier space. For example, if an attribute has a domain [a, z], this range is mapped to the whole identifier space (say between 0 and 1) and every node in the overlay takes the responsibility for a part of this range that falls between itself and its predecessor in the identifier space. For example, let us assume nodes X, Y, and Z are responsible for ranges [a, d], [d, f] and [f, g] in the attribute domain, which are in turn, mapped to [0, 0.1], [0.1, 0.3], and [0.3, 0.4] in the identifier space. Then, nodes that subscribe to any value in the first range, route towards node X, while nodes that subscribe to the values in the second or third ranges, route towards nodes Y or Z, respectively (Note, a node can subscribe to a range, which contains multiple rendezvous nodes, for example, a subscription for the range [b, e] will be routed to both nodes X and Y, each being responsible for a part of the subscription). Hence, multiple small trees are constructed for event delivery for this attribute. We further enhance the load balancing in Vinifera, by a novel technique that enables it to deal with non-uniform subscriptions and publications. Thus, we ensure an evenly distributed load, even in case the data in some regions of the identifier space is more popular or is published more frequently than the other regions.

The resulting balanced load in Vinifera is of critical importance, not only because it implies fairness and a higher resource utilization, but also, and most importantly, because it enables the system to function under massive data publications and tolerate failures.

We run extensive simulations to evaluate multiple aspects of the performance, namely scalability, fault tolerance, load balancing and congestion control. We compare Vinifera to a baseline system, constructed based on a state-of-the-art solution,
CHAPTER 7. VINIFERA

eFerry [100], which is a purely small-world overlay, oblivious to node subscriptions. Section 7.2 shows that this baseline solution is also conceptually equivalent to Ferry [100] and CAPS [101]. We show that, compared to the baseline system, Vinifera generates only one third of the traffic overhead, while the load is evenly distributed among the nodes and the delivery paths are up to four times shorter. We also show that Vinifera outperforms the baseline solution in the presence of churn, derived from real-world traces, and under intensive publications.

To summarize, our contribution is a genuinely scalable fault-tolerant multi-dimensional content-based publish/subscribe system with a bounded node degree requirement, a logarithmic worst-case bound on the delivery path length, and small and balanced load on the nodes. We achieve these properties by utilizing (i) an overlay topology that adapts to user subscriptions and exploits the similarity of subscriptions, in order to reduce the amount of traffic overhead that is generated in the network, (ii) constructing multiple efficient dissemination paths, instead of a rigid single tree structure, and (iii) a load balancing mechanism that enables the system to work under massive workloads.

7.2 Related work

As we briefly discussed in the introduction, there exist a number of solutions for building distributed content-based publish/subscribe systems [98] [99] [100] [101]. In this section we will have a closer look at these systems.

Meghdoot [98] exploits the idea of mapping each node subscription to a point in a $2d$ dimensional space, where $d$ is the number of attributes/dimensions in the subscription scheme. Then, a CAN [7] overlay is utilized for routing the messages. Although matching events against subscriptions can be nicely done in Meghdoot, the routing is not efficient, due to the inherent inefficiencies in CAN overlay. Moreover, node degree could grow linearly with the number of attributes. The load on the nodes is also very unbalanced, depending on where in the CAN overlay the node is positioned.

Sub-2-Sub [99] takes a completely different approach. It clusters the subscription space into multiple subspaces, where each subspace includes all and only the nodes that are subscribed to the whole subspace. From then on, each subspace is treated like a topic in a topic-based model. A ring is constructed over each subspace for disseminating the events inside that subspace. The problems are two fold: firstly, it is difficult to construct the subspaces, if subscriptions are complex. In Hyper [104], which is a non P2P solution for content-based publish/subscribe, it is proved that solving such a problem is NP-complete. The existence of churn in the P2P networks makes this problem even more challenging. Secondly, maintaining a ring per subspace implies that if a subscription is split into many subspaces, then the node has to join many overlays at the same time. Therefore, the node degree and maintenance cost could grow very large.

Ferry [100] is yet another approach to enable subscriptions over multiple attributes
by employing a structured overlay network. Every node hashes the attribute names and sends its subscription to a rendezvous (RV) node, which is responsible for one of the generated hash values, preferably to the closest one. All the subscriptions are then maintained at the RV nodes. Upon an event publication, the event is delivered to all the RV nodes and will be routed towards the subscribers, accordingly. The strong point in Ferry is that the node degree is bounded regardless of the number of attributes in the subscription scheme. However, since the nodes subscribe for the hash of the attribute names, the routing structure solely depends on the subscription scheme in the system. For example, if there is only one attribute in the model, then one RV node and one delivery tree will exist. Therefore, the load on the nodes will be extremely unbalanced. The RV node not only receives all the published events in the system, but also has to match each and every event against all the existing subscriptions, before relaying the received events. An effort to solve the problems in Ferry is presented in eFerry [102]. The approach is to use different combinations of several attributes, for subscription registration. The proposed mechanisms exhibits desirable load balancing properties only for the publish/subscribe system with extremely large number of attributes, while is still inefficient for the usual systems with one or few attributes.

Another solution, that also requires a bounded node degree, is CAPS [101]. Similar to Ferry, CAPS uses the rendezvous model for subscription installation and event delivery. The main difference is that instead of a single key per attribute, it generates a set of hash values for each subscription, and installs a node subscription in multiple RV nodes in the overlay. The matching is then performed at those RV nodes and events are forwarded along the overlay links from the RV nodes to the subscribers. The problem in CAPS is that a subscription may be translated into too many keys to be installed, and could potentially result in a high traffic in the network. Moreover, the matching is performed centrally at the RV nodes and there is no mechanism for load balancing.

Pyracanthus [105] is another solution, which uses order preserving hashing to enable range queries for content-based publish/subscribe. However, it has a high maintenance cost as it stores a node subscription in all the rendezvous nodes across all the attributes. Moreover, event publication is very costly in Pyracanthus, since the publisher requires to collect all node subscriptions from the rendezvous nodes for all the attributes. It then selects the matching subscribers among the collected subscriptions, and forwards the event to them.

BlueDove [106] is yet another solution for multi-dimensional content-based subscriptions, which is particularly designed for well-engineered environments like clouds. In such environments, data center servers are connected by high speed local networks, packet loss rate is very low, and servers stay on-line for long periods of time. There are also some related work on enabling range queries over P2P networks [28] [107]. Mercury [28], for example, supports multi-dimensional range-based searches, while it guarantees efficient routing and load balancing. Nevertheless, it does not provide all the necessary properties of a fully-fledged publish/subscribe system, since it lacks the mechanisms for installing user subscriptions and event delivery. Moreover, the
construction of the overlay in Mercury is oblivious to user interests. Another set of related work is focused on how to filter data content in the overlay networks [34] [35] [36]. However, these works are orthogonal to our work and can be complementary to Vinifera. In particular, we can utilize [34] on top of our dissemination trees in order to better filter out the published content. The focus in Vinifera is on building a topology that exploits user subscriptions to enable efficient data dissemination structures.

7.3 Architectural Model

Vinifera is a multi-layer solution, where each lower layer provides a service to its upper layers. The architectural model of Vinifera (Figure 7.1(a)) consists of three main layers:

Random overlay. On the bottom layer we have a random network, which we construct by a gossip-based peer sampling service [20], similar to Cyclon [26], NewsCast [25], or Gozar [22]. This service is periodically executed by all the nodes and provides each node with a random sample of the existing nodes in the overlay. This layer also enables nodes to propagate control information that are required by the upper layer. In particular, every node piggybacks its subscription information on the gossip messages that it sends out.

Vinifera overlay. The topology of this layer is constructed by capturing the existing subscription correlation in the system and clustering similar nodes together, using the same gossiping protocol. Moreover, we make this topology navigable by embedding it into an identifier space and enriching it by Small-World links following Kleinberg’s model [67]. The resulting topology allows efficient routing while preserving interest locality.

Vinifera Content management layer. This layer consists of several components that work together to manage the efficient delivery of the content. It exploits the navigability and the interest locality of the underlying layer by constructing a forest of dissemination structures based on RV routing. Because of the inherent interest locality property of the underlying overlay, each and every dissemination tree is expected to be small and efficient, involving mostly the interested nodes in the dissemination process. At the same time Vinifera trees are expected to be shorter as compared to the state-of-the-art.

7.4 Vinifera

Preliminaries

As we mentioned in the previous section, Vinifera is a gossip-based protocol, i.e., each nodes periodically exchanges some information with some other nodes in the overlay. This information includes the node profile, which contains the node identifier and the node subscription. The node identifier is selected uniformly at random
from a globally known identifier space. The subscription scheme includes \( n \) attributes from \( A_1 \) to \( A_n \), of any type, where attribute \( A_i \) could take values between \( v_{i_{\text{min}}} \) and \( v_{i_{\text{max}}} \). We map the range \( [v_{i_{\text{min}}}, v_{i_{\text{max}}}] \) to the entire node identifier space, by applying an order preserving hash function (OPHF) \([103]\) over the values that are valid for each attribute. An OPHF guarantees that if \( v > u \) then \( \text{OPHF}(v) > \text{OPHF}(u) \). For the sake of simplicity, from now on we refer to the hashed value \( \text{OPHF}(v) \), only as \( v \). Each node subscribes in the system by introducing a number of constraints over a subset of attributes. A constraint specifies either an exact value (equality) or a range of values for an attribute, and a subscription \( S \) is the conjunction of all such constraints. Figure 7.1(b) shows a system with two attributes \( A_1 \) and \( A_2 \) and three subscriptions: \( X \), \( Y \), and \( Z \). Subscriptions are shown by rectangles that specify the ranges over the two attributes. For example, subscription \( X \) is modeled as:

\[
S_X : A_1 \in [u, v] \land A_2 \in [s, t]
\]

A data item, or event, is a point in the attribute space, with exact values for all the attributes. An event matches a subscription, if each and every attribute value satisfies the corresponding constraint over that attribute. For example event \( e \) in Figure 7.1(b) matches subscription \( X \), but it does not match subscriptions \( Y \) and \( Z \).

**Components**

**Overlay Construction**

To enable nodes to select their neighbors (i.e., links or connections), based on their interest, identifier, or load, we employ a topology management protocol, inspired by T-Man \([16]\), on top of the peer sampling service provided by the underlying random overlay. Each node \( p \), maintains a routing table, i.e., a list of its neighbors, which it periodically exchanges with a neighbor, \( q \), chosen uniformly at random among the existing neighbors in the routing table. Node \( p \), then, merges its current routing table with the routing table of \( q \), together with a fresh list of the nodes, provided by the underlying peer sampling service. The resulting list becomes the candidate neighbors list for \( p \). Next, \( p \) selects a number of neighbors among the candidate neighbors and refreshes its current routing table. The same process will take place.
Algorithm 12 Select Primary Attribute

1: procedure selectPrimaryAttribute
2: for all \( A_i \) in self.S do \( \triangleright \) S represents node subscription
3: \hspace{1em} rank(\( A_i \)) \leftarrow 0 \( \triangleright \) initialize the rankings
4: end for
5: for all \( n \) in self.neighbors do
6: \hspace{1em} for all \( A_i \) in self.S do
7: \hspace{2em} if \( n.S.contain(\( A_i \)) \) then
8: \hspace{3em} self.C_i \leftarrow self.S.getC(\( A_i \)) \( \triangleright \) self constraint over \( A_i \)
9: \hspace{3em} n.C_i \leftarrow n.S.getC(\( A_i \)) \( \triangleright \) neighbor constraint over \( A_i \)
10: \hspace{3em} if overlapping(\( \text{self.C}_i \), \( \text{n.C}_i \)) \( \neq \) \( \emptyset \) then
11: \hspace{4em} rank(\( A_i \)) \leftarrow \text{rank}(\( A_i \)) + 1 \( \triangleright \) increment the rank of the attribute
12: \hspace{2em} end if
13: \hspace{1em} end if
14: end for
15: end for
16: \( A_p \leftarrow A_i \) where rank(\( A_i \)) is highest for all \( A_i \in \text{self.S} \)
17: end procedure

at node \( q \).

Every vinifera node selects three types of links. First, each node maintains two links to connect to the nodes that are closest to it in the node identifier space, one in each direction. These links are called ring links, because eventually these links shape up a ring structure in the overlay. The ring topology guarantees the existence of a path between any two nodes, and ensures the lookup consistency in the overlay, which is later required.

Next, to boost the routing efficiency, each node also selects some small-world links, based on the idea introduced by Kleinberg [67]. More precisely, node \( p \) selects node \( q \) as a small-world link, with a probability inversely proportional to the distance from \( p \) to \( q \) in the identifier space. It is proved that, having \( K_{sw} \) such neighbors enables a poly-logarithmic routing cost in the overlay \( (O(\frac{1}{K_{sw}} log^2 N)) \) [6].

Finally, links that are selected based on similarity of subscriptions are referred to as friend links. Every Vinifera node selects \( K_f \) friend links. These links connect together nodes with similar subscriptions. In a system with one attribute the similarity between two nodes \( p \) and \( q \) is captured by

\[
Utility(p, q) = \frac{S_p \cap S_q}{S_p \cup S_q} \quad \text{(Function I)}
\]

where, \( S_i \) contains the range(s) that node \( i \) has subscribed to. However, when there are more attributes, this approach is not readily applied. Instead, each node first selects one of the attributes, and then uses the mentioned utility function along that attribute only. As we will explain in Section 7.4, when an event is published in a system with multiple attributes, multiple copies of the event are propagated in the overlay, one for each attribute. Therefore, to guarantee the event delivery, it is enough if a node is efficiently located in a cluster associated with only one of the attributes. The clusterization, i.e., the friend links selection, is completed in two steps:
• A node first examines the subscriptions of its candidate neighbors to select an attribute, across which it has more neighbors with overlapping ranges. We refer to this attribute as the primary attribute for the node. Algorithm 12 illustrates how the primary attribute is selected. The basic idea is that a node assigns a rank to each of the attributes in its own subscription. The rank of an attribute is calculated by counting the number of neighbors with an overlapping interest on that attribute (Lines 10, 11). Finally, the attribute with the highest rank is selected as the primary attribute (Line 16).

• Next, the node uses the utility function (Function I) on the primary attribute and biases its neighbor selection towards selecting those nodes with higher rank as its friend links.

The combination of ring, small-world, and friend links results in a hybrid overlay, on top of which we build the data dissemination paths. We show, in the experiment section, that such a hybrid topology performs significantly better than a pure small-world overlay, as it not only reduces the unnecessary traffic in the network, but also improves the routing efficiency.

Routing

The basic routing or lookup in Vinifera is a greedy distance minimizing algorithm, i.e., at each step the lookup request is routed to a node which is closer to the destination. The destination of a lookup for value \( v \) is a node with the closest, but higher, identifier to \( v \), which we refer to as the rendezvous node (RV) for \( v \). In Vinifera, nodes can not only route towards exact values, e.g., \( v \), but also towards ranges of values, e.g., \([u, v]\) (Algorithm 13). To perform a range query, a node first applies an OPHF on the range boundary values. Then, a showering routing protocol [108] is executed, i.e., every node forwards the lookup request to as many nodes as it knows that fall into the range of the hashed values. Lines 15 to 17 in Algorithm 13 describe how the showering mechanism is performed. In case the node does not know any node in the requested range (line 18), it performs a simple greedy routing, i.e., it forwards the request to a node with closer, but not higher, identifier to the beginning of the range. The lookup ends at one or more consecutive RV nodes, each responsible for a part of the range.

Subscription Installation

Every node installs its subscription along the routing path to the rendezvous node(s) for the range over its primary attribute. We refer to this path (from the node itself to the rendezvous node(s)) as the installation path. Note that, we take advantage of the Vinifera overlay topology, by installing node subscriptions along the attribute for which they clustered more effectively, i.e., their primary attribute. As a result, nodes that have the same primary attribute, and a similar range of interest over
Algorithm 13 Range Query

1: procedure Lookup(requester, lookupRequest)
2: if requester ≠ self then
3: installNeighborSubscription(requester, lookupRequest)
4: end if
5: RVNodes ← NULL
6: if overlap(lookupRequest.range, [self, successor]) ≠ ⊘ then
7: RVNodes.add(successor)
8: end if
9: for all neighbor in self.neighbors do
10: if lookupRequest.range.includes(neighbor) then
11: RVNodes.add(neighbor)
12: end if
13: end for
14: if RVNodes ≠ NULL then
15: for all RV in RVNodes do
16: send lookup(self, lookupRequest) to RV
17: end for
18: else
19: nextHop ← findNextHop(lookupRequest.range.from)
20: send lookup(self, lookupRequest) to nextHop
21: end if
22: end procedure

that attribute, will lookup for the similar rendezvous nodes. Since these nodes
are very likely to be directly connected through friend links, they share most of
their routing path towards the rendezvous node(s), with high probability. This is
important, as it reduces the amount of traffic overhead transferred on the delivery
paths.

Figure 7.2(a) illustrates the lookup process with a single attribute. Assume node
Q wants to subscribe for the hashed values from 75 to 85. Among its neighbors,
it selects node R, which has the closest, but not higher, node identifier to the
requested range (Line 19 in Algorithm 13). The request is, therefore, sent from
Q to R. Node R forwards the request to its neighbor S, which falls into the requested
range (Line 10 in Algorithm 13). Node S takes the responsibility for the sub-range
[75, 78], and also forwards a request for the remaining sub-range to node T (Line
6 in Algorithm 13). Consequently, the installation path from the subscriber node
to the RV nodes is constructed (Path Q → R → S → T in Figure 7.2(a)).

Every node on the installation path maintains a table, called a Content Routing
Table or CRT for short. The CRTs are populated when the queries are forwarded
in the overlay. In our example, node R adds an entry to its CRT, for node Q
requesting range [75-85], while node S registers range [75, 85] for node R. Finally
node T registers a request from node S for the range [78-85]. In this example
we only have one attribute, and therefore CRTs, only include the requested range
for that single attribute. If there are more attributes, each entry associates the
requested range(s) with an attribute. Moreover, each entry of CRT contains the
complete subscription(s) of the requesting node over all the attributes. This field
will be used during event delivery process, described in Section 7.4.
Putting all these together, an entry of a CRT is a tuple in form of

\[
< \text{Ngh}, \text{Attr}, \text{Ranges}, \text{Subscriptions} >
\]

where \( \text{Ngh} \) indicates the requesting neighbor, \( \text{Attr} \) is the requested attribute, \( \text{Ranges} \) are the interested ranges over the requested attribute, and \( \text{Subscriptions} \) are the subscription requests, containing all the attributes, which are received through the requesting neighbor.

The subscription installation process is further equipped with an aggregation technique. That is, whenever possible, a node aggregates all the requests it receives from the same neighbor on the same attribute. This is usually referred to as subscription subsumption or covering in the literature. For example, in Figure 7.2(b) node \( P \) appears in the system and subscribes for the range \([80, 100]\). The closest neighbor of \( P \) to the requested range is node \( Q \). So \( P \) sends a request to \( Q \). As a result, \( Q \) installs this request in its CRT, and forwards it further to node \( R \). Since node \( R \) previously had an entry for \( Q \) in its CRT (for the range \([75-85]\)), it aggregates the two requests and modifies the entry for \( Q \) to range \([75-100]\). Now \( R \) knows two nodes, \( S \) and \( T \), in the requested range. So it showers the request to both of them, by sending a request for range \([75-78]\) to \( S \) and the remaining part of the range to \( T \). When this new request is further forwarded in the overlay, nodes \( S \) and \( T \) similarly update their CRTs with the range \([75, 78]\) and the aggregated range \([78-100]\), respectively. Then, node \( T \) forwards the request further to node \( U \), which is also a rendezvous node for a part of the requested range.

As mentioned previously, thanks to the employed clustering technique, nodes with similar subscriptions on the same primary attribute are grouped together. When these nodes install their subscriptions, they initiate a routing towards the same or close-by rendezvous nodes. Therefore, the installation tree is mostly shared between such nodes, thus, the requests can be effectively aggregated. This results in smaller CRTs, as well as less traffic overhead in the overlay. Smaller CRTs not only reduce the maintenance cost of the trees, but also simplify the matching process. Note that, the subscription installation is a periodic process, and therefore if a node fails or changes its subscription, it does not send any more request for the previously requested range, and therefore, the already installed rows in CRTs further
ahead, can de-aggregate or be removed completely. This ensures that the quality of CRTs are constantly maintained.

**Event Delivery**

Any node in Vinifera can publish events. As mentioned previously, an event is a piece of data that has an exact value for each attribute. Therefore, in a system with \( n \) attributes, an event is associated with \( n \) rendezvous nodes, one for each attribute. When a node publishes an event \( e\{v_1, v_2, \ldots, v_n\} \), it sends one copy of the event to each of the \( n \) relevant rendezvous nodes, i.e., \( RV(v_1), RV(v_2), \ldots, RV(v_n) \), which are responsible for the values assigned to each of the attributes. This is done by initiating a simple rendezvous routing for each attribute. Then, \( n \) delivery trees for the event are constructed on the fly, by following the links on the reverse installation paths from the rendezvous nodes to the subscriber nodes, using the node CRTs. Note that each matching subscriber is registered in only one of the delivery trees, i.e., the one that corresponds to its primary attribute. So, it does not receive redundant messages from multiple trees.

The delivery trees are constructed as follows. Each rendezvous node, matches the event against the subscriptions that are registered in its CRT, and sends the event only to the neighbors with matching subscriptions. Note that the matching is performed on the whole registered subscriptions, that is, if the event does not match the registered subscription of a node on any of the attributes (primary or not), the event will not be forwarded further to that node. Likewise, every node on the path performs such a matching process and forwards the event further if it matches the registered request, until it reaches the interested subscribers. By this approach, the matching process will be carried out as the event goes down to the subscribers, while every node maintains partial information about the other nodes. In essence, we distribute the load of matching events against subscriptions between the nodes that are on the installation path. At every step, those branches that are not interested in the event are pruned and the event is forwarded only down the paths that hold some interested node(s).

**Load Balancing**

Due to the prevalent skewed subscription patterns in the real world, the use of an OPHF inevitably results in a non-uniform identifier distribution and thus, an unbalanced load on the nodes. More precisely, some regions in the identifier space might be very popular with huge number of corresponding events, while some other regions might not be popular at all. So, the nodes who happen to fall into the popular regions may have to deal with huge number of requests. For example, if an attribute in the subscription scheme is temperature in a city, then the published events are most likely in the range \([-10, +30]\), probably a few around this range, and almost no event in less than \(-30\) or over \(+50\). Hence, node that have an identifier between OPHF\((-10)\) and OPHF\(+30\) are likely to be highly loaded, while
the rest of the nodes are under loaded. To alleviate this problem, we utilize an idea, inspired by [109], for adapting the node identifiers to the existing load in the network. The idea is that Vinifera nodes may change their identifiers along the ring, while their connections remain intact. In other word, nodes change their identifier to an identifier between themselves and their successor. The new identifier is assigned to the node to halve the load on the successor. Since nodes do not change their neighbors upon change of the identifier, they can easily inform their neighbors of the new identifier, when they send the next gossip message to the neighbors.

For our system we define a measure of load as follows. Every node counts the number of events that it receives as a rendezvous node, without having any interest in them. Whenever the node sends its gossip message to its predecessor, it piggybacks its current load on the message. In order to prevent perturbation of the node identifiers, we define a threshold $\beta$ for load imbalance between a node and its successor. When the difference of load between the two nodes exceeds the threshold $\beta$, the proceeding node changes its identifier to a value closer to its successor. Then the two nodes update their load to the average of their current loads. We show in the evaluation section that by employing this mechanism, Vinifera nodes can adapt to even very skewed user subscriptions.

Maintenance

In general, P2P networks are subject to churn, i.e., nodes join or leave the system continuously and concurrently, and network capacity changes. Therefore, it is essential to design P2P systems that tolerate failures. When a node fails in Vinifera, all the layers take the required actions to deal with that failure. The random overlay at the bottom, which is a gossip-based peer sampling service, is inherently failure tolerant. More details on fault tolerance in such protocols can be found in [20].

In the Vinifera overlay layer, failure handling is done similarly to the random overlay. As we described in Section 7.4, nodes periodically send their profile to their neighbors. This profile message, therefore, serves as a heartbeat message and enables the nodes to detect the failure of their direct neighbors. When a node fails, its direct neighbors that detect the failure, remove the entries for the failed neighbor from their partial view. When these nodes exchange their views with other nodes in the system, the contacted nodes will also receive the updated information and remove the stale entries, accordingly. Therefore, the information about the failed nodes, is propagated in the overlay by taking advantage of the ongoing gossiping protocol.

In the content management overlay, we need to ensure that CRTs are always updated and do not contain stale entries, i.e., when a node fails, we should remove its subscription from all the CRTs along the installation path. Note that, every entry in the CRT has an expiry timestamp. If a node does not receive a new subscription request from its neighbor, it will automatically remove the request from its CRT. Normally, in each gossip round requests are resent and therefore maintained
7.5 Evaluation

We implemented Vinifera in Peersim [110], a discrete event simulator for P2P applications. Through extensive simulations with a hybrid of cycle based and event based models, we evaluated the performance of Vinifera, while comparing it against a baseline system, inspired by the state-of-the-art techniques such as Ferry [100], eFerry [102], and CAPS [101]. That is, the baseline system is a pure small-world overlay, thus, requires a bounded node degree and guarantees a bounded routing time, but it is oblivious to node subscription, with no load balancing mechanism. In the lack of real-world traces, we synthetically generated user subscriptions, using a Zipf-like distribution over the attribute space [111]. Unless otherwise mentioned, the network size is 1000.

Topology Configuration

In this experiment, we investigate the choice of values for parameters $K_{sw}$, and $K_f$, which define the number of small-world links and friend links, respectively. The total number of links per node is set to 10, among which two are dedicated to ring links and the rest are used for small-world and friend links, i.e., $K_{sw} + K_f = 8$. Figures 7.3a and 7.3b show the traffic overhead and average delivery path length in the CRTs. When a node fails, however, the first node met on the installation path, which has been directly connected to the failed node, detects the failure and removes the subscription of the failed node from its CRT. Therefore, it never again forwards the requests for that subscription, i.e., the next node on the relay path, will not receive the request for that subscription, thus, removes the entry from its CRT. Note that, if an entry in the CRT is the result of an aggregation, i.e., a node $A$ received two requests from the same neighbor $B$ for some overlapping ranges, and part of this aggregated range concerned a node $C$ that is failed now, only the part that corresponded to $C$ will be de-aggregated, as node $B$ will send only one of the requests to node $A$, thereafter.
of Vinifera and the baseline system for two subscription models. Since the baseline system does not have any friend links, its topology does not change across the X-axis. Zero friend link in Vinifera generates a pure small-world topology, which is oblivious to node subscriptions, just like the baseline system. Hence, at this point both systems have the same topology and that is why with random subscriptions, the improvement in the traffic overhead in Vinifera is negligible. However at the same point, the average path length in Vinifera is decreased. This is due to the existence of many short delivery trees in Vinifera, as opposed to a single long delivery tree in Ferry. With skewed subscriptions, we can also improve the traffic overhead from over 32% to 22%. By adding more friend links, we take advantage of the subscription similarities and significantly improve both metrics at the same time. With 8 friends and skewed subscriptions, for example, the traffic overhead drops from 32% to less than 10%, while the average path length is decreased from around 7 to 3. This proves the huge potential of exploiting user subscription correlations, common in real-world scenarios. In the rest of our experiments, we set $K_{sw}$ to 0 and $K_f$ to $\log(N) - 2$, where $N$ is the number of nodes in the network. However, for the applications that require an upper bound guarantee on the number of delivery hops, we can add more small-world links.

**Scalability**

To measure the scalability of Vinifera, we performed experiments with different number of nodes, as well as, different number of attributes. Figure 7.4a shows the traffic overhead of both systems. The performance of both systems is almost the same for different network sizes. However, the traffic overhead in the baseline system is more than 80% for random subscriptions, whereas it is reduced to 60% in Vinifera. Note that random subscriptions bring up the worst case scenario, for nodes can not effectively benefit from our clustering technique, due to the lack of correlation between user subscriptions. However, it is shown that a significant subscription correlation exists in real-world application [76] [77]. When the user subscriptions are skewed, the traffic overhead in the baseline system drops to nearly half, while Vinifera reduces the traffic to almost one sixth of that of the random subscriptions. This shows that our data dissemination overlay is remarkably benefiting from the utilized clustering technique. Figure 7.4b shows the average delivery path length of both systems in terms of hop counts. Here again, the number of hops in Vinifera is nearly one third of that of the baseline system. However, the path length is slightly bigger with skewed subscriptions, because the overlay topology is clustered. With the random subscription model, however, the overlay better resembles a random network, thus, we observe a reduced path length.

Next, we observe the behavior of both systems when the subscription model includes more attributes. We have designed Vinifera to work with any dimensionality. Because of the lack of real-world traces we had to decide on the number of dimensions ourselves while carrying out the experiments. To be consistent (as well as being able to easily compare) with the existing state-of-the-art solutions
we used most of the parameters (including dimensionality) from the related work papers. As we observed, most of the related work had reported results with 2, 3, or 5 attributes. We also show the results with up to 5 attributes. In higher dimensions, Vinifera still exhibits consistent results. Nevertheless, with our randomly generated events, the fraction of matching events drops sharply, thus, the measured values become insignificant, making the system hard to evaluate. This is due to a phenomenon, known as the curse of dimensionality in the literature [112].
7.5. EVALUATION

However, since in real life the generated data is not uniformly spread in the data space and there exists a correlation between user subscriptions and the generated data [111] [76] [77], Vinifera is expected to function effectively in higher dimensions under realistic workloads.

Figure 7.4c shows when the subscription model is random, the traffic overhead of the baseline system remains at around 80%. This overhead starts from around 58% in Vinifera, but increases in higher dimensions, because in a random subscription scheme there exists very little similarities to be exploited, and therefore, the primary attribute is practically a random attribute for each node. As a result, the installation paths, and thus, the delivery trees are scattered in the overlay and nodes cannot effectively cooperate in data dissemination. However, this overhead is still less than that of the baseline system. With the skewed subscription model, both systems behave significantly better. The overall improvement is again due to the skewed event publication in the system. However, the improvement in the baseline system with more attributes is because instead of having one single tree, more delivery trees are constructed, one for each attribute. Each node joins one of these trees, as a subscriber, and does not receive the events that are forwarded on other trees, unless it is a relay node in those trees. However, in Vinifera nodes with similar subscriptions are grouped together, and the delivery tree is shared between these nodes. Thus, the overall number of uninterested node on the delivery trees is reduced, thus, the traffic overhead drops to nearly one third of that of the baseline system.

The average delivery path length of the two systems are shown in Figure 7.4d. The baseline system delivers the events slightly faster when there are more attributes. However, the average delivery path length in Vinifera is still by far better than the baseline system, even with 5 attributes in the subscription model, thanks to the utilized clustering technique. As soon as an event reaches a cluster of nodes with matching subscriptions, it is propagated inside that cluster very quickly.

We conclude that although both systems can accommodate any number of attributes in the subscription scheme, Vinifera exhibits a significantly better performance than the baseline system, especially in the presence of skewed subscriptions.

Load Balancing

To explore how the load is distributed among the node in Vinifera versus the baseline system, we plot the cumulative distribution of load on the nodes, for 1, 2, and 3 attributes, and report the results in Figure 7.4e. Although with more attributes, the load distribution is slightly degraded in Vinifera, it is still significantly better than the baseline system and the load on any Vinifera node never exceeds 30%. More precisely, over 95% of the nodes has a load less than 20%, even with 3 attributes, whereas, 10% of the baseline system nodes suffer from over 60% load in the system, while nearly 40% of the nodes have zero load. Figure 7.4e shows that the load in the baseline system is extremely unbalanced. This is because nodes up on the delivery tree are highly stressed, while leaf nodes are just receiving the service for free.
There are nodes with even nearly 100% load (the rendezvous nodes), which can significantly harm the performance of the system as soon as they stop functioning correctly, due to congestion or failure. This problem prevents the baseline system to work under real-life scenarios where node and network failures are inevitable, while Vinifera can still function without having any imminent bottleneck.

**Workload**

In this section we examine if the systems can function under different workloads, i.e., under different event publication rates. To model the congestion, we assume that every node in the system can handle a bounded number of messages, $X$, in every round, and if it receives more events it will simply drop them. Then, we increase the number of events that are published in the system to up to five times $X$.

Figure 7.4f shows that the hit ratio in the baseline system significantly drops as soon as the publication rate passes the $X$ threshold. Whereas, Vinifera survives even under high event publication rates. This is due to the fact that the baseline system relies on very few nodes to propagate the event in the system (only the intermediary nodes in the delivery tree). Therefore, under a high publication rate, those nodes
7.6. CONCLUSIONS

become highly overloaded and start dropping the messages. When a node in the
tree drops a message, all its descendant nodes fail to receive the message. On the
other hand, in Vinifera load is almost evenly distributed among the nodes. Thus,
the nodes do not have to drop the messages due to the excessive load.

**Fault Tolerance**

To evaluate the performance of Vinifera in the presence of failures, we used real-
world churn traces [113], that were obtained by monitoring a set of 4000 nodes
participating in the Skype superpeer network for one month beginning September
12, 2005. In Figures 7.5a to 7.5c the x-axis shows the time, while the y-axis on
the right shows the network size. The black solid line in the three graphs shows
how the network size changes over time. Figure 7.5a shows the hit ratio of the two
systems with random and skewed subscriptions. Although the hit ratio of Vinifera
slightly decreases in the flash crowds, i.e., around time 100 h., when a large number
of nodes join the system concurrently, the system recovers quickly and the hit ratio
goes back to and remain at 100%, even in the presence of further joins and failures.
In contrast, the hit ratio in the baseline system is highly affected by churn, due to
the fragile structure of a single delivery tree. When this tree is broken, the baseline
system can not repair it quickly enough to catch up with further event deliveries.
When no more node joins or fails, the baseline system is potentially able to repair
the dissemination tree. However, as we see in this real-world trace, this hardly
happens.

Figure 7.5b shows the traffic overhead in both systems. The traffic overhead in
Vinifera is one forth compared to the baseline system for both random and skewed
subscriptions. Note that the reduced traffic overhead in the baseline system is
because it fails to deliver the events to all the nodes. We also observe, in Figure 7.5c,
that Vinifera is takes a four times shorter delivery path compared to the baseline
system, in the presence of churn. Here again, we should take into account that
in the baseline system some nodes are not receiving the events, and the measured
values for the baseline system only include the nodes that received the events.

7.6 Conclusions

We introduced Vinifera, a P2P content-based publish/subscribe system that enables
users to subscribe for the information they are willing to receive, without having
to rely on any single authority or central server. We employed a gossip-based tech-
nique to construct a topology that not only resembles a small-world network, but
also connects the nodes with similar subscriptions together. On top of this hybrid
overlay, we utilized a rendezvous routing mechanism to propagate node subscrip-
tions in the overlay. Together with an order preserving hashing technique and an
efficient showering algorithm we enabled range queries, and at the same time, we
employed a load balancing technique to deal with the potential non-uniform user
subscriptions. The combination of all these techniques are seamlessly integrated
within a single gossiping layer, thus keeping Vinifera simple, lightweight and robust.
Our hybrid publish/subscribe system exhibited superior performance against the state-of-the-art techniques, effectively without the need to trade-off or degrade any important properties of the system. The overlay topology autonomously adapts to user subscriptions and is highly resilient to the dynamism in the network. The generated traffic overhead and the average delivery path length are simultaneously kept low, while only a bounded node degree is required and no global knowledge at any point is assumed.
Chapter 8

JabeJa: A Distributed Algorithm for Balanced Graph Partitioning

Fatemeh Rahimian, Amir H. Payberah, Sarunas Girdzijauskas, Mark Jelasity and Seif Haridi

In the 7th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO), Philadelphia, USA, September 2013.
8.1. INTRODUCTION

**JabeJa: A Distributed Algorithm for Balanced Graph Partitioning**

Fatemeh Rahimian†‡, Amir H. Payberah†
Sarunas Girdzijauskas‡, Mark Jelasity and Seif Haridi†

†Swedish Institute of Computer Science (SICS)
‡KTH - Royal Institute of Technology
{amir, fatemeh, seif}@sics.se
sarunascg@kth.se

Abstract

Balanced graph partitioning is a well known NP-complete problem with a wide range of applications. These applications include many large-scale distributed problems including the optimal storage of large sets of graph-structured data over several hosts—a key problem in today’s Cloud infrastructure. However, in very large-scale distributed scenarios, state-of-the-art algorithms are not directly applicable, because they typically involve frequent global operations over the entire graph. In this paper, we propose a fully distributed algorithm, called Ja-be-Ja, that uses local search and simulated annealing techniques for graph partitioning. The algorithm is massively parallel: there is no central coordination, each node is processed independently, and only the direct neighbors of the node, and a small subset of random nodes in the graph need to be known locally. Strict synchronization is not required. These features allow Ja-be-Ja to be easily adapted to any distributed graph-processing system from data centers to fully distributed networks. We perform a thorough experimental analysis, which shows that the minimal edge-cut value achieved by Ja-be-Ja is comparable to state-of-the-art centralized algorithms such as Metis. In particular, on large social networks Ja-be-Ja outperforms Metis, which makes Ja-be-Ja—a bottom-up, self-organizing algorithm—a highly competitive practical solution for graph partitioning.

8.1 Introduction

Every day, petabytes of data are generated and processed in on-line social networking services. Some of this data can be modeled as a graph, in which nodes represent users and edges represent the relationship between them. Similarly, search engines manage very large amounts of data to capture and analyze the structure of the Internet. Likewise, this data can be modeled as a graph, with websites as nodes and the hyperlinks between them as edges. One important problem related to graph-structured data processing is partitioning: extremely large scale graphs must be distributed to hosts in such a way, that most of the adjacent edges are stored on the same host [1].
Finding good partitions is a well-known and well-studied problem in graph theory [114]. The graph partitioning problem, sometimes referred to as the \textit{min-cut problem}, is formulated as dividing a graph into a predefined number of components, such that the number of edges between different components is small. A variant of this problem is the \textit{balanced} or uniform graph partitioning problem, where it is also important that the components hold an equal number of nodes. The examples of important applications include biological networks, circuit design, parallel programming, load balancing, graph databases and on-line social network analysis. The motivation for graph partitioning depends on the application. A good partitioning can be used to minimize communication cost, to balance load, or to identify densely connected clusters. Figures 8.1a and 8.1b are examples of a poor and a good partitioning of a graph, respectively.

In this paper, we focus on processing extremely large-scale graphs, e.g., user relationship and interaction graphs from online social networking services such as Facebook or Twitter, resulting in graphs with billions of nodes and hundreds of billions of edges. The very large scale of the graphs we target poses a major challenge. Although a very large number of algorithms are known for graph partitioning [44–46, 48–50, 115, 116], including parallel ones, most of the techniques involved assume a form of cheap random access to the entire graph. In contrast to this, large scale graphs do not fit into the main memory of a single computer, in fact, they often do not fit on a single local file system either. Worse still, the graph can be fully distributed as well, with only very few nodes hosted on a single computer.

We provide a distributed balanced graph partitioning algorithm, which does not require any global knowledge of the graph topology. That is, we do not have cheap
access to the entire graph and we have to process it only with partial information.

Our solution, called JA-BE-JA, is a decentralized local search algorithm. Each node of the graph is a processing unit, with local information about its neighboring nodes, and a small subset of random nodes in the graph, which it acquires by purely local interactions. Initially, every node selects a random partition, and over time nodes swap their partitions to increase the number of neighbors they have in the same partition as themselves.

Our algorithm is uniquely designed to deal with extremely large distributed graphs. The algorithm achieves this through its locality, simplicity and lack of synchronization requirements, which enables it to be adapted easily to graph processing frameworks such as Pregel [2] or GraphLab [1]. Furthermore, JA-BE-JA can be applied on fully distributed graphs, where each network node represents a single graph vertex.

To evaluate JA-BE-JA, we use multiple datasets of different characteristics, including a few synthetically generated graphs, some graphs that are well-known in the graph partitioning community [52], and some sampled graphs from Facebook [53] and Twitter [54]. We first investigate the impact of different heuristics on the resulting partitioning of the input graphs, and then compare JA-BE-JA to METIS [45], a well-known centralized solution. We show that, although JA-BE-JA does not have cheap random access to the graph data, it can work as good as, and sometimes even better than, a centralized solution. In particular, for large graphs that represent real-world social network structures, such as Facebook and Twitter, JA-BE-JA outperforms METIS [45].

In the next section we define the exact problem that we are targeting, together with the boundary requirements of the potential applications. In Section 8.3 we study the related work of graph partitioning. Then, in Section 8.4 we explain JA-BE-JA in detail, and evaluate it in Section 8.5. Finally, in Section 8.6 we conclude the work.

8.2 Problem statement

The problem that we address in this paper is distributed balanced k-way graph partitioning. In this section we formulate the optimization problem and describe our assumptions about the system we operate in.

Balanced k-way graph partitioning

We are given an undirected graph $G = (V, E)$, where $V$ is the set of nodes (vertices) and $E$ is the set of edges. A $k$-way partitioning divides $V$ into $k$ subsets. Intuitively, in a good partitioning the number of edges that cross the boundaries of components is minimized. This is referred to as the min-cut problem in graph theory. Balanced (uniform) partitioning refers to the problem of partitioning the graph into equal-sized components. The equal size constraint can be softened by requiring that the partition sizes differ only by a factor of a small $\epsilon$. 
A $k$-way partitioning can be given with the help of a partition function $\pi : V \rightarrow \{1, \ldots, k\}$ that assigns a color to each node. Hence, $\pi(p)$, or $\pi_p$ for short, refers to the color of node $p$. Nodes with the same color form a partition. We denote the set of neighbors of node $p$ by $N_p$, and define $N_p(c)$ as the set of neighbors of $p$ that have color $c$:

$$N_p(c) = \{q \in N_p : \pi_q = c\}$$ (8.1)

The number of neighbors of node $p$ is denoted by $d_p$, and $d_p(c) = |N_p(c)|$ is the number of neighbors of $p$ with color $c$. We define the energy of the system as the number of edges between nodes with different colors (equivalent to edge-cut). Accordingly, the energy of a node is the number of its neighbors with a different color, and the energy of the graph is the sum of the energy of the nodes:

$$E(G, \pi) = \frac{1}{2} \sum_{p \in V} (d_p - d_p(\pi_p)),$$ (8.2)

where we divide the sum by two since the sum counts each edge twice. Now we can formulate the balanced optimization problem: find the optimal partitioning $\pi^*$ such that

$$\pi^* = \arg \min_{\pi} E(G, \pi)$$ (8.3)

$$s.t. \quad |V(c_1)| = |V(c_2)|, \forall c_1, c_2 \in \{1, \ldots, k\}$$ (8.4)

where $V(c)$ is the set of nodes with color $c$.

**Data distribution model**

We assume that the nodes of the graph are processed periodically and asynchronously, where each node only has access to the state of its immediate neighbors and a small set of random nodes in the graph. The nodes could be placed either on an independent host each, or processed in separate threads in a distributed framework. This model, which we refer to as the *one-host-one-node* model, is appropriate for frameworks like GraphLab [1] or Pregel [2], Google’s distributed framework for processing very large graphs. It can also be used in peer-to-peer overlays, where each node is an independent computer. In both cases, no shared memory is required. Nodes communicate only through messages over edges of the graph, and each message adds to the communication overhead.

The algorithm can take advantage of the case, when a computer hosts more than one graph node. We call this the *one-host-multiple-nodes* model. Here, nodes on the same host can benefit from a shared memory on that host. For example, if a node exchanges some information with other nodes on the same host, the communication cost is negligible. However, information exchange across hosts is costly and constitutes the main body of the communication overhead. This model is interesting for data centers or cloud environments, where each computer can emulate thousands of nodes at the same time.
8.3 Related Work

Graph Partitioning

There exist quite a few works that address the $k$-way balanced graph partitioning problem in a centralized model. Also, there are partitioning algorithms that have a distributed model similar to that of JA-BE-JA, but do not compute a predefined number of balanced partitions. Here, we briefly overview some of these algorithms. To the best of our knowledge, JA-BE-JA is the first algorithm that fills in the gap between these two sets of algorithms and can produce balanced partitions in a completely distributed model.

Balanced Graph Partitioning Algorithms

Metis [45] is a widely known and successful algorithm based on Multilevel Graph Partitioning (MGP) [114]. MGP generally works in three phases: (i) a sequence of smaller graphs are produced from the original graph, by iteratively contracting edges and unifying nodes. This is repeated until the number of nodes in the coarsened graph is small enough to perform an inexpensive partitioning, (ii) the smallest graph is partitioned, and (iii) the partitions are propagated back through a sequence of un-contracting nodes and edges.

Note that the best partition for the coarsened graph may not be optimal for the uncoarsened original graph, thus, the third phase also includes some local refinements to improve the cut size as the edges are un-contracted. Therefore, the MGP approach is usually coupled with other heuristics for local refinement, e.g., Kernighan-Lin (KL) algorithm [117]. Metis combines several heuristics during its coarsening, partitioning, and uncoarsening phases to improve the cut size. It also uses a greedy refinement method, which was found to be significantly faster than the original MGP algorithm.

There are many other algorithms based on MGP. For example, Soper et al. [118] proposed an algorithm that combined a Genetic Algorithm (GA) technique with MPG. In [118] crossover and mutation operators are used to compute edge biases, which yield hints for the underlying multilevel graph partitioner. Chardaire et al. [119] also proposed a meta-heuristic, which can be viewed as a GA without selection. Benlic et al. [120] provided a perturbation-based iterated tabu search procedure for partition refinement of each coarsened graph. KAFFPA [116] is another MGP algorithm using local improvement algorithms that are based on flows and localized searches.

In order to speedup the partitioning process for very large-scale graphs, designing algorithms that can be parallelized is inevitable. pARMetis [46] is the parallel version of Metis that improves the partitioning time, but at the cost of lower quality partitions. KAFFPAE [50] is also a parallelized MGP algorithm, which produces even better partitions compared to its non-parallel ancestor KAFFPA [116]. Moreover, Talbi et al. proposed a parallel graph partitioning technique in [121] based
on parallel GA [122]. Although these algorithms can produce the final partitioning faster, they require access to the entire graph at all times, which renders them very expensive for large graphs that can not fit into the memory of a single computer.

### Distributed Graph Partitioning Algorithms

Apart from Ja-be-Ja, there exist some other algorithms that operate based on partial information. The decentralized nature of these algorithms enables them to process very large graphs. For example, DiDiC [123] is a distributed diffusion-based algorithm that eliminates all the global operations for assigning nodes to partitions. Also, Cdc [124], which adopts some ideas from the diffusion-based models, is particularly designed for peer-to-peer networks. However, unlike Ja-be-Ja, these solutions may produce partitions of drastically different sizes. We initially carried out experiments with DiDiC for our problem, however had to abandon it since we observed that it tends to find good-shaped partitions rather than balanced ones, and therefore, the number and size of yielded partitions can not be controlled, as it depends on the topology of the input graph.

### 8.4 Solution

We propose Ja-be-Ja, a distributed heuristic algorithm for the balanced $k$-way graph partitioning problem.

#### The basic idea

Recall, that we defined the energy of the system as the number of edges between nodes with different colors, and the energy of a node is the number of its neighbors with a different color. The basic idea is to initialize colors uniformly at random, and then to apply heuristic local search to push the configuration towards lower energy states (min-cut).

The local search operator is executed by all the graph nodes in parallel: each node attempts to change its color to the most dominant color among its neighbors. However, in order to preserve the size of the partitions, the nodes cannot change their color independently. Instead, they only swap their color with one another. Each node iteratively selects another node from either its neighbors or a random sample, and investigates the pair-wise utility of a color exchange. If the color exchange decreases the energy then the two nodes swap their colors. Otherwise, they preserve their colors.

When applying local search, the key problem is to ensure that the algorithm does not get stuck in a local optimum. For this purpose, we employ the simulated annealing technique [125] as we describe below. Later, in the evaluation section

\[1\text{JA-BE-JA means swap in Persian.} \]
(Section 8.5), we show the impact of this technique on the quality of the final partitioning.

Note that, since no color is added to/removed from the graph, the distribution of colors is preserved during the course of optimization. Hence, if the initial random coloring of the graph is uniform, we will have balanced partitions at each step. We stress that this is a heuristic algorithm, so it cannot be proven (or, in fact, expected) that the globally minimal energy value is achieved. Exact algorithms are not feasible since the problem is NP-complete, so we cannot compute the minimum edge-cut in a reasonable time, even with a centralized solution and a complete knowledge of the graph. In Section 8.5, however, we compare our results with the best known partitioning over a number of benchmark problem instances.

Swapping: the local search operator

Firstly, a node selects a set of candidate nodes for swapping. We consider three possible ways of selecting the candidate set:

- **Local** (*L*): every node considers its directly connected nodes (neighbors) as candidates for color exchange.

- **Random** (*R*): every node selects a uniform random sample of the nodes in the graph. Note that there exist multiple techniques for taking a uniform sample of a given graph at a low cost [15, 22, 26, 126–128].

- **Hybrid** (*H*): in this policy first the immediate neighbor nodes are selected (i.e., the local policy). If this selection fails to improve the pair-wise utility, the node is given another chance for improvement, by letting it to select nodes from its random sample (i.e., the random policy).

Secondly, a node needs to define how to select the swap *partner*. The partner of a node *p* is the node that *p* chooses among its candidates to exchange its color with. To decide if two nodes should swap their colors, we require: (i) a function to measure the pair-wise utility of a color exchange, and (ii) a policy for escaping local optima.
In order to minimize the edge-cut of the partitioning, we try to maximize $d_p(\pi_p)$ for all nodes $p$ in the graph, which only requires local information at each node. Two nodes $p$ and $q$ with colors $\pi_p$ and $\pi_q$, respectively, exchange their colors only if this exchange decreases their energy (increases the number of neighbors with a similar color to that of the node):

$$d_p(\pi_q)^\alpha + d_q(\pi_p)^\alpha > d_p(\pi_p)^\alpha + d_q(\pi_q)^\alpha$$  \hspace{1cm} (8.5)$$

where $\alpha$ is a parameter of the energy function. If $\alpha = 1$, a color exchange is accepted if it increases the total number of edges with the same color at two ends. For example, color exchange for nodes $p$ and $q$ in Figure 8.2a is accepted, as the nodes change from a state with 1 and 0 neighbors of a similar color, to 1 and 3 such neighbors, respectively. However, nodes $u$ and $v$ in Figure 8.2b, each in a state with 2 neighbors of a similar color, do not exchange their colors, if $\alpha = 1$, because $2 + 2 \neq 1 + 3$. However, if $\alpha > 1$, then nodes $u$ and $v$ will exchange their colors. Although, this exchange does not directly reduce the total edge-cut of the graph, it increases the probability of future color exchanges for the two yellow nodes, currently in the neighborhood of node $v$. In section 8.5 we evaluate the effect of the parameter $\alpha$.

To avoid becoming stuck in a local optimum, we use the well-known Simulated Annealing (SA) technique [125]. We introduce a temperature ($T$) and decrease it over time, similar to the cooling process in [125]. The updated decision criterion becomes

$$\left(d_p(\pi_q)^\alpha + d_q(\pi_p)^\alpha\right) \times T > d_p(\pi_p)^\alpha + d_q(\pi_q)^\alpha.$$  \hspace{1cm} (8.6)$$

As a result, in the beginning we might move in a direction that degrades the energy function, i.e., nodes exchange their color even if the edge-cut is increased. Over time, however, we take more conservative steps and do not allow those exchanges that result in a higher edge-cut. The two parameters of the SA process are (i) $T_0$, the initial temperature, which is greater than or equal to one, and (ii) $\delta$, that determines the speed of the cooling process. The temperature in round $r$ is calculated as $T_r = T_{r-1} - \delta$. When the temperature reaches the lower bound 1, it is not decreased anymore. From then on, the decision procedure falls back on using equation (8.5).

We also use a multi-start search [125], by running the algorithm many times, starting from different initial states. Note that this technique is applied in a distributed way. More precisely, after each run, nodes use a gossip-based aggregation method [15] to calculate the edge-cut in the graph. If the new edge-cut is smaller than the previous one, they update the best solution found so far by storing the new edge-cut value together with the current local color.

Ja-Be-Ja

JA-BE-JA combines the two aforementioned components: the sampling policy and swapping technique. Algorithm 14 presents the core of JA-BE-JA. As shown in
Algorithm 14 Ja-be-Ja Algorithm.

Require: Any node $p$ in the graph has the following methods:

- $\text{getNeighbors}()$: returns $p$’s neighbors.
- $\text{getSample}()$: returns a uniform sample of all the nodes.
- $\text{getDegree}(c)$: returns the number of $p$’s neighbors that have color $c$.

1: //Sample and Swap algorithm at node $p$
2: procedure SAMPLEANDSWAP
3: partner ← $\text{FindPartner}(p.\text{getNeighbors}(), T_r)$
4: if partner = null then
5: partner ← $\text{FindPartner}(p.\text{getSample}(), T_r)$
6: end if
7: if partner $\neq$ null then
8: color exchange handshake between $p$ and partner
9: end if
10: $T_r ← T_r - \delta$
11: if $T_r < 1$ then
12: $T_r ← 1$
13: end if
14: end procedure

15: //Find the best node as swap partner for node $p$
16: function FINDPARTNER(Node[] nodes, float $T_r$)
17: highest ← 0
18: bestPartner ← null
19: for $q \in$ nodes do
20: $d_{pp} ← p.\text{getDegree}(p.color)$
21: $d_{qq} ← q.\text{getDegree}(q.color)$
22: old ← $d_{pp} + d_{qq}$
23: $d_{pq} ← p.\text{getDegree}(q.color)$
24: $d_{qp} ← q.\text{getDegree}(p.color)$
25: new ← $d_{pq} + d_{qp}$
26: if (new $\times T_r > old) \land (new > highest)$ then
27: bestPartner ← $q$
28: highest ← new
29: end if
30: end for
31: return bestPartner
32: end function

method $\text{SampleAndSwap}$, we use the hybrid heuristic for node selection, which first
tries the local policy (line 3), and if it fails it follows the random policy (line 5). Method \textbf{FindPartner} shows how the partner is selected. We calculate the two sides of equation (8.5) in lines 20 – 25, and in line 26, we compare these computed values. Here, the current temperature, $T_r$, biases the comparison towards selecting new states (in the initial rounds).

Note that the actual swapping operation is implemented as an optimistic transaction, the details of which are not included in the algorithm listing to avoid distraction from the core algorithm. The actual swap is done after the two nodes perform a handshake and agree on the swap. This is necessary, because the deciding node might have outdated information about the partner node. During the handshake, the initiating node sends a swap request to the partner node, along with all the information that the partner node needs to verify the swap utility: the current color ($\pi_p$), the partner’s color ($\pi_{\text{partner}}$), the number of neighbors with the same color ($d_p(\pi_p)$), and the number of neighbors with the color of the partner node ($d_p(\pi_{\text{partner}})$). If the verification succeeds, the partner node replies with an acknowledgment (ACK) message and the swap takes place. Otherwise, a negative acknowledgment message (NACK) is sent and the two nodes preserve their previous colors. These sample and swap processes are periodically repeated by all the nodes, in parallel, and when no more swaps take place in the graph, the algorithm has converged.

In the one-host-multiple-nodes model, the only change required to the core algorithm is to give preference to local host swaps. That is, if there are several nodes as potential partners for a swap, the node selects the one that is located on the local host, if there is such a candidate. Note that in this model not each and every node requires to maintain a random view for itself. Instead, the host can maintain a large enough sample of the graph to be used as a source of samples for all hosted nodes. In Section 8.5, we study the trade-off between communication overhead and the edge-cut with and without considering the locality.

\section*{Generalizations of Ja-Be-Ja}

So far, we have discussed the case when the graph links are not weighted and the partition sizes are equal. However, JA-BE-JA is not limited to these cases. In this section, we briefly describe how it can deal with weighted graphs and produce arbitrary pre-defined partition sizes.

\textbf{Weighted graphs.} In real world applications links are often weighted. For example, in a graph database some operations are performed more frequently, thus, some links are accessed more often [129]. In order to prioritize such links when partitioning the graph, we change the definition of $d_p$, such that, instead of just counting the number of neighboring nodes with the same color, we sum the weights of these links:

$$d_p(c) = \sum_{q \in N_p(c)} w(p, q)$$  \hspace{1cm} (8.7)
where \( w(p, q) \) is the weight of the edge between \( p \) and \( q \).

**Arbitrary partition sizes.** For example, assume we want to split the data over two machines that are not equally powerful. If the first machine has twice as many resources than the second one, we need a 2-way partitioning with one component being twice as large as the other. To do that, we can initialize the graph partitioning with a biased distribution. For example, if nodes initially choose randomly between two partitions \( c_1 \) and \( c_2 \), such that \( c_1 \) is twice as likely to be chosen, then the final partitioning will have a partition \( c_1 \), which is twice as big. This is true for any distribution of interest, as JA-BE-JA is guaranteed to preserve the initial distribution of colors.

### 8.5 Experimental evaluation

We implemented JA-BE-JA on PeerSim [110], a discrete event simulator for building P2P protocols. First, we investigate the impact of different heuristics and parameters on different types of graphs. Then, we conduct an extensive experimental evaluation to compare the performance of JA-BE-JA to (i) METIS [45], a well-known efficient centralized solution, and (ii) the best known available results from the Walshaw benchmark [52] for several graphs. Unless stated otherwise, we compute a 4-way partitioning of the input graph with initial temperature \( T_0 = 2 \), the temperature is reduced by \( \delta = 0.003 \) in each step until it reaches value 1, and parameter \( \alpha \) is set to 2.

**Metrics**

Although the most important metric for graph partitioning is edge-cut (or energy), there are a number of studies [130] that show that the edge-cut alone is not enough to measure the partitioning quality. Several metrics are, therefore, defined and used in the literature [48, 49], among which we selected the following ones in our evaluations:

- **edge-cut**: the number of inter-partition edges, as given in Formula 8.2, i.e., \( E(G, \pi) \).
- **swaps**: the number of swaps that take place between different hosts during run-time (that is, swaps between graph nodes stored on the same host are not counted).
- **data migration**: the number of nodes that need to be migrated from their initial partition to their final partition.

While the edge-cut is a quality metric for partitioning, the number of swaps defines the cost of the algorithm. Moreover, the data migration metric makes sense only in the one-host-multiple-nodes model, where some graph nodes have to
Datasets

We have used three types of graphs: (i) two synthetically generated graphs, (ii) several graphs from Walshaw archive [52], and (iii) sampled graphs from two well-known social networks: Twitter [54] and Facebook [53]. These graphs are listed in Table 8.1.

**Table 8.1: Datasets**

<table>
<thead>
<tr>
<th>Dataset</th>
<th></th>
<th>V</th>
<th></th>
<th></th>
<th>E</th>
<th></th>
<th>Type</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synth-WS</td>
<td>1000</td>
<td>4147</td>
<td>Synth.</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Synth-SF</td>
<td>1000</td>
<td>7936</td>
<td>Synth.</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add20</td>
<td>2395</td>
<td>7462</td>
<td>Walshaw</td>
<td>52</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>data</td>
<td>2851</td>
<td>15093</td>
<td>Walshaw</td>
<td>52</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3elt</td>
<td>4720</td>
<td>13722</td>
<td>Walshaw</td>
<td>52</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4elt</td>
<td>15606</td>
<td>45878</td>
<td>Walshaw</td>
<td>52</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vibrobox</td>
<td>12328</td>
<td>165250</td>
<td>Walshaw</td>
<td>52</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Twitter</td>
<td>2731</td>
<td>164629</td>
<td>Social</td>
<td>54</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Facebook</td>
<td>63731</td>
<td>817090</td>
<td>Social</td>
<td>53</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

migrate from one physical machine to another after finding the final partitioning. If the graph nodes that are initially located at a given host get the same initial color, then this metric is given by the number of nodes that end up with a different color by the time the algorithm has converged.

**Synthetic Graphs.** We generated two different graphs synthetically. The first one is based on the Watts-Strogatz model [131], with 1000 nodes and average degree 8 per node. First, a lattice is constructed and then some edges are rewired with probability 0.02. We refer to this graph as Synth-WS. The second graph, Synth-SF, is an implementation of the Barabasi-Albert model [132] of growing scale free networks. This graph also includes 1000 nodes with an average degree of 16. Both graphs are undirected and there are no parallel edges either.

**The Walshaw Archive.** The Walshaw archive [52] consists of the best partitioning found to date for a set of graphs, and reports the partitioning algorithms that achieved those best results. This archive, which has been active since the year 2000, includes the results from most of the major graph partitioning software packages, and is kept updated regularly by receiving new results from the researchers in this field. For our experiments, we have chosen graphs add20, data, 3elt, 4elt, and vibrobox, which are the small and medium size graphs in the archive, listed in Table 8.1.
8.5. EXPERIMENTAL EVALUATION

The Social Network Graphs. Since social network graphs are one of the main targets of our partitioning algorithm, we investigate the performance of JA-BE-JA on two sampled datasets, which represent the social network graphs of Twitter and Facebook.

We sampled our Twitter graph from the follower network of 2.4 million Twitter users [54]. There are several known approaches for producing an unbiased sample of a very large social network, such that the sample has similar graph properties to those of the original graph. We used an approach discussed in [84] sampling nearly 10000 nodes by performing multiple breadth first searches (BFS). We also used a sample graph of Facebook, which is made available by Viswanath et. al. [53]. This data is collected by crawling the New Orleans regional network during December 29th, 2008 and January 3rd, 2009, and includes those users who had a publicly accessible profile in the network. The data, however, is anonymized.
The impact of the sampling policies

In this section, we study the effect of different sampling heuristics on the edge-cut. These heuristics were introduced in Section 8.4 and are denoted by $L$, $R$, and $H$. Here, we evaluated the one-node-one-host model, and to take uniform random samples of the graph we applied Newscast [15, 25] in our implementation. As shown in Table 8.2, all heuristics significantly reduce the initial edge-cut that belongs to a random partitioning. Even with heuristic $L$, which only requires the information about direct neighbors of each node, the edge-cut is reduced to 30% of the initial number for the Facebook graph. The random selection policy, i.e., heuristic $R$, works even better than local ($L$) for all the graphs, as it is less likely to get stuck in a local optimum. The best result for most graphs, however, is achieved with the combination of $L$ and $R$: the hybrid heuristic ($H$).
Table 8.2: Different sampling heuristics, with $\alpha = 2$ and SA.

<table>
<thead>
<tr>
<th>Graph</th>
<th>initial</th>
<th>$L$</th>
<th>$R$</th>
<th>$H$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synth-WS</td>
<td>3127</td>
<td>1051</td>
<td>600</td>
<td>221</td>
</tr>
<tr>
<td>Synth-SF</td>
<td>5934</td>
<td>4571</td>
<td><strong>4151</strong></td>
<td>4169</td>
</tr>
<tr>
<td>add20</td>
<td>5601</td>
<td>3241</td>
<td>1446</td>
<td>1206</td>
</tr>
<tr>
<td>data</td>
<td>11326</td>
<td>3975</td>
<td>1583</td>
<td>775</td>
</tr>
<tr>
<td>3elt</td>
<td>10315</td>
<td>4292</td>
<td>1815</td>
<td>390</td>
</tr>
<tr>
<td>4elt</td>
<td>34418</td>
<td>14304</td>
<td>6315</td>
<td>1424</td>
</tr>
<tr>
<td>vibrobox</td>
<td>123931</td>
<td>42914</td>
<td><strong>22865</strong></td>
<td>23174</td>
</tr>
<tr>
<td>Twitter</td>
<td>123683</td>
<td>45568</td>
<td>41079</td>
<td><strong>40775</strong></td>
</tr>
<tr>
<td>Facebook</td>
<td>612585</td>
<td>181661</td>
<td>119551</td>
<td><strong>117844</strong></td>
</tr>
</tbody>
</table>

Table 8.3: Tuning $\alpha$, with Hybrid sampling and SA.

<table>
<thead>
<tr>
<th>Graph</th>
<th>initial</th>
<th>$\alpha = 1$</th>
<th>$\alpha = 2$</th>
<th>$\alpha = 3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synth-WS</td>
<td>3127</td>
<td>265</td>
<td><strong>221</strong></td>
<td>290</td>
</tr>
<tr>
<td>Synth-SF</td>
<td>5934</td>
<td>4190</td>
<td><strong>4169</strong></td>
<td>4215</td>
</tr>
<tr>
<td>add20</td>
<td>5601</td>
<td><strong>1206</strong></td>
<td><strong>1206</strong></td>
<td>1420</td>
</tr>
<tr>
<td>data</td>
<td>11326</td>
<td><strong>618</strong></td>
<td>775</td>
<td>1241</td>
</tr>
<tr>
<td>3elt</td>
<td>10315</td>
<td>601</td>
<td><strong>390</strong></td>
<td>1106</td>
</tr>
<tr>
<td>4elt</td>
<td>34418</td>
<td>1473</td>
<td><strong>1424</strong></td>
<td>2704</td>
</tr>
<tr>
<td>vibrobox</td>
<td>123931</td>
<td>23802</td>
<td><strong>23174</strong></td>
<td>25602</td>
</tr>
<tr>
<td>Twitter</td>
<td>123683</td>
<td><strong>40775</strong></td>
<td>41040</td>
<td>41247</td>
</tr>
<tr>
<td>Facebook</td>
<td>612585</td>
<td>124328</td>
<td><strong>117844</strong></td>
<td>133920</td>
</tr>
</tbody>
</table>

The impact of the swapping policies

In these experiments, we study the effect of the parameters that define the swapping policies, introduced in Section 8.4. We investigate the impact of the parameters on the final edge-cut, as well as on the number of swaps. Table 8.3 contains the edge-cut values achieved with different values of $\alpha$, a parameter of the swapping condition in equation (8.5). The setting $\alpha = 2$ gives the best result for most of the graphs. In Section 8.4 we explained why $\alpha$ is better to be greater than 1. In this experiment, we observe that if $\alpha$ is set too high, nodes might overestimate the value of a swap and end up in an inferior state.

Table 8.4 lists the edge-cut with and without simulated annealing (SA). In the simulations without SA, we set $T_0 = 1$, which is the lowest allowed temperature in our case (see equation (8.6)). Although the improvements due to SA might be minor for some graphs, for other graphs with various local optima SA can lead to a much smaller edge-cut. We also ran several experiments to investigate the effect of $T_0$ and observed that $T_0 = 2$ gives the best results in most cases. These experiments
are not reported due to lack of space.

The other parameter of the simulated annealing technique is $\delta$, the speed of the cooling process. We investigate the impact of $\delta$ on the edge-cut and on the number of swaps. Figure 8.3 shows the results as a function of different values for $\delta$. The higher $\delta$ is, the higher the edge-cut is (Y1-axis) and the smaller the number of swaps is (Y2-axis). In other words, $\delta$ represents a trade-off between the number of swaps and the quality of the partitioning (edge-cut). Note that a higher number of swaps means both a longer convergence time and more communication overhead. For example, for $\delta = 0.003$, it takes around 334 rounds for the temperature to decrease from 2 to 1, and in just very few rounds after reaching the temperature of 1 the algorithm converges. Interestingly, the social network graphs are very robust to $\delta$ in terms of the edge-cut value, so in the case of highly clustered graphs the best choice seems to be a relatively fast cooling schedule.

**Locality**

Here, we investigate the evolution of the edge-cut, the number of swaps, and the number of migrations over time, assuming the one-host-multiple-nodes model. Recall, that swaps between nodes within the same host are not counted. We assume there are four hosts in the systems, where each host gets a random subset of nodes initially. They run the algorithm to find a better partitioning by repeating the sample and swap steps periodically, until no more swaps occur (convergence). As shown in Figure 8.4, in both models, the algorithm converges to the final partitioning in round 350, that is, shortly after the temperature reaches 1. We also observe that the convergence time is mainly dependent on the parameters of the simulated annealing process, and so it can be controlled by the initial temperature $T_0$ and the cooling schedule parameter $\delta$.

Although (as we have seen) we can achieve a much lower number of swaps in Twitter and Facebook graphs with higher values of $\delta$ without sacrificing the solution quality (Figures 8.3c and 8.3d), we have performed these experiments with the same setting of $\delta = 0.003$ for all the graphs. As shown in Figure 8.4b, locally biased swapping results in relatively more inter-host swaps over the 3elt graph. Fortunately, in the rest of the graphs—that include the practically interesting social network samples as well—we can see the opposite (and more favorable) trend, namely that JA-BE-JA achieves the same edge-cut with much fewer inter-host swaps. We speculate that this is due to the fact that in the latter group of graphs there are various different partitionings of the graph with a similar edge-cut value, thus, local swaps will be more likely to be good enough.

When the goal is to re-arrange the graph, data is not actually moved before the algorithm has converged to the final partitioning. Instead, on a given host, all nodes are initialized with the same color. During run-time, only the color labels are exchanged. The color of a node may change several times before convergence. When the algorithm converges, each data item (node) is migrated from its initial partition to its final partition indicated by its color.
8.5. EXPERIMENTAL EVALUATION

Table 8.4: The impact of SA on the edge-cut ($\alpha = 2$).

<table>
<thead>
<tr>
<th>Graph</th>
<th>initial</th>
<th>$H$</th>
<th>$H + SA$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synth-WS</td>
<td>3127</td>
<td>503</td>
<td>221</td>
</tr>
<tr>
<td>Synth-SF</td>
<td>5934</td>
<td>4258</td>
<td>4169</td>
</tr>
<tr>
<td>add20</td>
<td>5601</td>
<td>1600</td>
<td>1206</td>
</tr>
<tr>
<td>data</td>
<td>11326</td>
<td>1375</td>
<td>775</td>
</tr>
<tr>
<td>3elt</td>
<td>10315</td>
<td>1635</td>
<td>390</td>
</tr>
<tr>
<td>4elt</td>
<td>34418</td>
<td>6240</td>
<td>1424</td>
</tr>
<tr>
<td>vibrobox</td>
<td>123931</td>
<td>26870</td>
<td>23174</td>
</tr>
<tr>
<td>Twitter</td>
<td>123683</td>
<td>41087</td>
<td>41040</td>
</tr>
<tr>
<td>Facebook</td>
<td>612585</td>
<td>152670</td>
<td>117844</td>
</tr>
</tbody>
</table>

Table 8.5: The number of nodes that need to migrate.

| graph     | $|V|$ | $|mig|$ |
|-----------|-----|------|
| Synth-WS  | 1000| 720  |
| add20     | 2395| 1740 |
| 3elt      | 4720| 3436 |
| Twitter   | 2731| 2000 |
| Facebook  | 63731| 47555|

Note that migration could be optimized given the final partitioning, but we simply assume that nodes with a color different from the original color will migrate. Table 8.5 shows the number of data items that need to be migrated after the convergence of the algorithm. As expected, this number constitutes nearly 75% of the nodes for a 4-way partitioning. This is because each node initially selects one out of four partitions uniformly at random, and the probability that it is not moved to a different partition is only 25%. Equivalently, 25% of the nodes stay in their initial partition and the remaining 75% have to migrate.

Comparison With the State-of-the-Art

In this section, we compare JA-BE-JA to METIS [45] on all the input graphs. We also compare these results to the best known solutions for the graphs from the Walshaw benchmark [52]. Table 8.6 shows the edge-cut of the final 4-way partitioning. As shown, for some graphs, METIS produces better results, and for some others JA-BE-JA works better. However, the advantage of JA-BE-JA is that it does not require all the graph data at once, and therefore, it is more practical when processing very large graphs.

Next, we investigate the performance of the algorithms, in terms of edge-cut, when the number of the required partitions grows. Figure 8.5 shows the resulting
Table 8.6: JA-BE-JA vs. METIS vs. the best known edge-cut.

<table>
<thead>
<tr>
<th>Graph</th>
<th>JA-BE-JA</th>
<th>METIS</th>
<th>Best known edge-cut</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synth-WS</td>
<td>221</td>
<td>210</td>
<td>-</td>
</tr>
<tr>
<td>Synth-SF</td>
<td>4169</td>
<td>4279</td>
<td>-</td>
</tr>
<tr>
<td>add20</td>
<td>1206</td>
<td>1276</td>
<td>1159 (Probe [119])</td>
</tr>
<tr>
<td>data</td>
<td>775</td>
<td>452</td>
<td>382 (MMA02 [133])</td>
</tr>
<tr>
<td>3elt</td>
<td>390</td>
<td>224</td>
<td>201 (JE [118])</td>
</tr>
<tr>
<td>4elt</td>
<td>1424</td>
<td>374</td>
<td>326 (NW [134])</td>
</tr>
<tr>
<td>vibrobox</td>
<td>23174</td>
<td>22526</td>
<td>19098 (MMA02 [133])</td>
</tr>
<tr>
<td>Twitter</td>
<td>41040</td>
<td>65737</td>
<td>-</td>
</tr>
<tr>
<td>Facebook</td>
<td>117844</td>
<td>117996</td>
<td>-</td>
</tr>
</tbody>
</table>

(a) add20 graph.  
(b) 3elt graph.  
(c) Twitter graph.  
(d) Facebook graph.

Figure 8.5: JA-BE-JA vs. METIS scalability in different graphs.
edge-cut of JA-BE-JA versus METIS for 2 to 64 partitions. Naturally, when there are more partitions in the graph, the edge-cut will also grow. However, as shown in most of the graphs (except for 3elt), JA-BE-JA finds a better partitioning compared to METIS, when the number of partitions grows. In particular, JA-BE-JA outperforms METIS in the social network graphs. For example, as shown in Figure 8.5d the edge-cut in METIS is nearly 20,000 more than JA-BE-JA. Note, unlike METIS, JA-BE-JA does not make use of any global information or operation over the entire graph.

8.6 Conclusions

We provided an algorithm that, to the best of our knowledge, is the first distributed algorithm for balanced graph partitioning that does not require any global knowledge. To compute the partitioning, nodes of the graph require only some local information and perform only local operations. Therefore, the entire graph does not need to be loaded into memory, and the algorithm can run in parallel on as many computers as available. We showed that our algorithm can achieve a quality partitioning, as good as a centralized algorithm. We also studied the trade-off between the quality of the partitioning versus the cost of it, in terms of the number of swaps during the run-time of the algorithm.

Acknowledgment

M. Jelasity was supported by the Bolyai Scholarship of the Hungarian Academy of Sciences. This work was partially supported by the European Union and the European Social Fund through project FuturICT.hu (grant no.: TAMOP-4.2.2.C-11/1/KONV-2012-0013).
Chapter 9

Distributed Vertex-Cut Partitioning

Fatemeh Rahimian, Amir H. Payberah, Sarunas Girdzijauskas, and Seif Haridi

In the 14th IFIP international conference on Distributed Applications and Interoperable Systems (DAIS’14).
Distributed Vertex-Cut Partitioning

Fatemeh Rahimian†‡, Amir H. Payberah†
Sarunas Girdzijauskas‡ and Seif Haridi†

†Swedish Institute of Computer Science (SICS)
‡KTH - Royal Institute of Technology
{amir, fatemeh, seif}@sics.se
sarunasmg@kth.se

Abstract

Graph processing has become an integral part of big data analytics. With the ever increasing size of the graphs, one needs to partition them into smaller clusters, which can be managed and processed more easily on multiple machines in a distributed fashion. While there exist numerous solutions for edge-cut partitioning of graphs, very little effort has been made for vertex-cut partitioning. This is in spite of the fact that vertex-cuts are proved significantly more effective than edge-cuts for processing most real world graphs. In this paper we present Ja-be-Ja-vc, a parallel and distributed algorithm for vertex-cut partitioning of large graphs. In a nutshell, Ja-be-Ja-vc is a local search algorithm that iteratively improves upon an initial random assignment of edges to partitions. We propose several heuristics for this optimization and study their impact on the final partitioning. Moreover, we employ simulated annealing technique to escape local optima. We evaluate our solution on various graphs and with variety of settings, and compare it against two state-of-the-art solutions. We show that Ja-be-Ja-vc outperforms the existing solutions in that it not only creates partitions of any requested size, but also requires a vertex-cut that is better than its counterparts and more than 70% better than random partitioning.

9.1 Introduction

A wide variety of real-world data can be naturally described as graphs. Take for instance communication networks, social networks, biological networks, etc. With the ever increasing size of such networks, it is crucial to exploit the natural connectedness of their data in order to store and process them efficiently. Hence, we are now observing an upsurge in the development of distributed and parallel graph processing tools and techniques. Since the size of the graphs can grow very large, sometimes we have to partition them into multiple smaller clusters that can be processed efficiently in parallel. Unlike the conventional parallel data processing, parallel graph processing requires each vertex or edge to be processed in the context of its neighborhood. Therefore, it is important to maintain the locality of information while partitioning the graph across multiple (virtual) machines. It is also
important to produce equal size partitions that distribute the computational load evenly between clusters.

Graph partitioning is a well known NP-Complete problem in graph theory. In its classical form, graph partitioning usually refers to edge-cut partitioning, that is, to divide vertices of a graph into disjoint clusters of nearly equal size, while the number of edges that span separated clusters is minimum. However, tools that utilize edge-cut partitioning do not achieve good performance on real-world graphs (which are mostly power-law graphs) [38–40], mainly due to unbalanced number of edges in each cluster. In contrast, both theory [41] and practice [42, 43] prove that power-law graphs can be efficiently processed in parallel if vertex-cuts are used.

A vertex-cut partitioning divides edges of a graph into equal size clusters. The vertices that hold the endpoints of an edge are also placed in the same cluster as the edge itself. However, the vertices are not unique across clusters and might have to be replicated (cut), due to the distribution of their edges across different clusters. A good vertex-cut is one that requires minimum number of replicas. Figure 9.1 illustrate the difference between these two types of partitioning.

While there exist numerous approximate solutions for edge-cut partitioning, very little work has investigated vertex-cut partitioning. Figure 9.2 shows a graph with three different vertex-cut partitionings. The graph edges are partitioned into two clusters. Two colors, yellow and blue, are representing these two partitions. Vertices that have edges of one color only, are also colored accordingly, and the vertices that have to be replicated are colored white. A very naïve solution is to randomly assign edges to partitions. As shown in Figure 9.2a, nearly all the vertices have edges of different colors, thus, they have to be replicated in both partitions. Figure 9.2b illustrates what happens if we use an edge-cut partitioner, and then assign the cut edges to one of the partitions randomly. As shown, the vertex-cut improves significantly. However, the number of edges in the partitions is very unbalanced. What we desire is depicted in Figure 9.2c, where the vertex-cut is kept as low as possible, while the size of the partitions, with respect to the number of edges, is balanced.

An alternative solution for vertex-cut partitioning of a graph $G$ is to transform it to its corresponding line graph $L(G)$ (where $L(G)$ represents the adjacencies between edges of $G$) and then use an edge-cut partitioning algorithm. However, in most real-world graphs the number of edges are orders of magnitude higher than the number of vertices, thus, the line graph often has a significantly higher number of vertices. Consequently, the complexity of the partitioning could grow
9.1. INTRODUCTION

drastically. It is, therefore, necessary to devise algorithms that performs the vertex-cut partitioning on the original graph.

In this paper, we present a distributed vertex-cut partitioning algorithm, called JA-BE-JA-VC, based on local search optimization, which is mainly inspired by our previous work for edge-cut partitioning [56]. The algorithm starts with a random assignment of edges to partitions. For simplicity we represent each partition with a distinct color. Over time, vertices exchange information (about the color of their edges) with each other and try to locally reduce the vertex-cut, by negotiating over the assignment of their edges to partitions. Every vertex attempts to assign all its edges to the same partition (same color), because this means the vertex does not have to be cut. If this case is not possible due to the contention between neighboring vertices, then the vertex tries to have the minimum number of distinct assignments. Two vertices will decide to exchange the colors of their candidate edges, if the vertex-cut can be reduced. Otherwise, the edge colors are preserved.

The aforementioned heuristic is likely to get stuck in local optima due to initial random partitioning and the nature of the problem, which is NP-Complete. Thus, we employ the well-known simulated annealing technique [125] to escape local optima and find a better vertex-cut. Note, JA-BE-JA-VC will always maintain the initial distribution of partition sizes. That is, if the initialization is uniformly random, the partition sizes are expected to be balanced. If we require to have partitions of a different distribution, e.g., one partition twice as big as the others, then we only need to change the initialization step to produce the required distribution.

We observe through experiments, that JA-BE-JA-VC produces quality partitions on several large graphs and scales well with varying number of partitions. We also study the trade-off between the vertex-cut and the computation cost, in terms of the number of iterations to compute the partitioning. Finally, we compare JA-BE-JA-VC to the state-of-the-art vertex-cut partitioner [55], as well as the state-of-the-art edge-cut partitioner [56], and study their existing the trade-offs. We show that JA-BE-JA-VC consistently outperforms [56] and [55] on producing requested size partitions, while not sacrificing the vertex-cut. Even for varying number of partitions, JA-BE-JA-VC always reduces the vertex-cut to lower than 30% and down

Figure 9.2: Vertex-cut partitioning into two clusters. The color of each edge/vertex represents the partition it belongs to. The white vertices belong to both partitions.

(a) A random vertex-cut. Edges are evenly distributed, but nearly all the vertices have to be replicated.

(b) Partitioning based on edge-cut and then assigning the cut edges randomly to one partition. Only one vertex is replicated, but edges are not evenly distributed.

(c) A good vertex-cut partitioning. Three vertices are replicated and the number of edges in the two partitions are balanced.
CHAPTER 9. DENA

to 10% for some graphs.

9.2 Problem statement

We are given an undirected graph $G = (V, E)$, where $V$ is the set of vertices and $E$ is the set of edges. A $k$-way balanced vertex-cut partitioning divides the set of edges $E$ into $k$ subsets of equal size, where $k$ is an input parameter. Each partition also has a subset of vertices that hold at least one of the edges in that partition. However, vertices are not unique across partitions, that is, some vertices may appear in more than one partition, due to the distribution of their edges across several partitions. A good edge partitioning strives to minimize the number of vertices that belong to more than one partition.

A $k$-way balanced vertex-cut partitioning can be given with the help of a partition function $\pi : E \to \{1, \ldots, k\}$ that assigns a color to each edge. Hence, $\pi(e)$, or $\pi_e$ for short, refers to the color of edge $e$. Edges with the same color form a partition. We denote the set of edges that are connected to vertex $p$ by $E_p$. Accordingly, $E_p(c)$ indicates the subset of edges of $p$ that have color $c$:

$$E_p(c) = \{e \in E_p : \pi_e = c\} \quad (9.1)$$

We refer to $|E_p(c)|$ as the cardinality of color $c$ at vertex $p$. Then, the energy of a vertex $p$ is shown with $\gamma(p, \pi)$ and it is defined as the number of different colors that has a cardinality greater than zero.

$$\gamma(p, \pi) = \sum_{|E_p(c)| > 0} 1, \forall \ c \in \{1, \ldots, k\} \quad (9.2)$$

In other words, the energy of a vertex $p$ for a partition function $\pi$ is the number of different colors that are assigned to the edges of $p$, which is equivalent to the number of required replicas (vertex-cut) for $p$. The energy of the graph is sum of the energy of all its vertices:

$$\Gamma(G, \pi) = \sum_{p \in V} \gamma(p, \pi) \quad (9.3)$$

Now we can formulate an optimization problem as follows: find the optimal partitioning $\pi^*$ such that:

$$\pi^* = \arg \min_{\pi} \Gamma(G, \pi) \quad (9.4)$$

$$s.t. \ |E(c_1)| = |E(c_2)|, \forall \ c_1, c_2 \in \{1, \ldots, k\} \quad (9.5)$$

where $|E(c)|$ is the number of edges with color $c$. Note, in all practical cases the second condition is relaxed, such that it requires partitions of approximately equal size. This is important, because the number of edges of the graph is not necessarily a multiple of $k$. Therefore, throughout this paper, we address the relaxed version of the problem.
9.3. Solution

In order to partition the edges of a given graph, we use an approach inspired by Ja-be-Ja [56], our previous work on edge-cut partitioning. Our algorithm, called JA-BE-JA-VC, is vertex-centric and fully distributed, and no central point with a global knowledge is required. Vertices of the graph execute the algorithm independently and iteratively. In this algorithm, initially a random color is assigned to each edge of the graph. This is equivalent to a random assignment of edges to partitions. Then we allow vertices to exchange the color of their edges, provided that the exchange leads to a better cut of the graph. In the initialization step we can control the required partition size distribution. If edges are initially assigned to partitions uniformly at random, then size of the partitions is expected to be equal. We could also use any other distribution for initial edge assignment, and JA-BE-JA-VC guarantees to preserve this distribution during the course of optimization.

The optimization step is illustrated in Algorithm 15. In each iteration, first a vertex checks whether or not it is internal. An internal vertex is a vertex that is surrounded with the edges of the same color (i.e., \( \gamma(p, \pi) = 1 \)). If the vertex is internal, it does not need to perform any optimization and waits for its turn in the next round. Otherwise, the vertex proceeds with the following three steps: (i) edge selection, (ii) partner selection, and (iii) swap. Each of these steps could be realized by means of various policies. Here we explain a few possible policies for these steps separately.

**Edge Selection**

In this step a vertex selects one of its edges for color exchange. We consider two policies for edge selection: (i) random and (ii) greedy. In the random policy, a vertex chooses one edge of its edges randomly. Although random selection is very straightforward, it will not lead our local search in the right direction. Consider, for example, a vertex with a majority of edges with blue color and just very few red edges. If this vertex selects a random edge for a color exchange, it is more likely that the vertex selects a blue edge (because it has a majority of blue edges), and such selection is not in the interest of the vertex. Whereas, if the vertex selects an edge with red color, it will have a higher chance of unifying the color of its edges. With a greedy policy, a vertex selects one of its edges, e.g., \( e \), which has a color with minimum cardinality:

\[
\exists e \in E_p(c^*), \quad c^* = \arg \min_c |E_p(c)|
\]

Since random policy is ineffective in our optimization process, we only consider the greedy edge selection in our experiments. Algorithm 16 describes how an edge is selected with this policy.
Algorithm 15 Optimization sketch

1: procedure RUN()
2:     if self.isInternal() is not TRUE then
3:         selfEdge ← self.selectEdge()           ▷ Select an edge (Algorithm 16)
4:         candidates ← self.selectCandidates() ▷ Select a list of candidate vertices (Algorithm 17)
5:             for all partner in candidates do   ▷ Look for a swap partner among the candidates
6:                 if partner is not internal then
7:                     if policy is DominantColor then
8:                         selfColor ← self.getDominantColor()
9:                         partnerColor ← partner.getDominantColor()
10:                        if selfColor ≠ partnerColor && partner.hasEdge(selfColor) then
11:                            partnerEdge ← partner.selectEdge(selfColor)
12:                            swapColor(selfEdge, partnerEdge)
13:                            break
14:                     end if
15:                 else                              ▷ If the policy is based on Edge Utility
16:                     partnerEdge ← partner.selectEdge()
17:                     if swapUtility(selfEdge, partnerEdge) > 0 then
18:                         swapColor(selfEdge, partnerEdge)
19:                         break
20:                 end if
21:             end if
22:         end if
23:     end for
24: end if
25: end procedure

Algorithm 16 Edge Selection

1: procedure SELECTEDGE(color)
2:     if color is null then
3:         color ← self.getColorWithMinCardinality()
4:     end if
5:     edges ← self.getEdges(color)
6:     return edges.getRandomElement()
7: end procedure

Partner Selection

In this step a vertex selects a subset of other vertices from the graph as candidates for a color exchange. A vertex considers two sets of vertices for partner selection:
9.3. SOLUTION

Algorithm 17 Partner Selection

1: procedure selectCandidates()
2: \( \text{candidates} \leftarrow \text{self.getNeighbours().getSubset()} \) \(\triangleright\) a subset of direct neighbors
3: \( \text{candidates}.add(\text{getRandomVertices()} ) \) \(\triangleright\) a subset of random vertices from the graph
4: return candidates
5: end procedure

Algorithm 18 Calculate Swap Utility

1: procedure swapUtility(edge1, edge2)
2: \( c_1 \leftarrow \text{edge1.getColor()} \)
3: \( c_2 \leftarrow \text{edge2.getColor()} \)
4: \( u_{1c1} \leftarrow \text{getEdgeValue(edge1.src, edge1.dest, c1)}; \) \(\triangleright\) utility of edge1 before swap
5: \( u_{2c2} \leftarrow \text{getEdgeValue(edge2.src, edge2.dest, c2)}; \) \(\triangleright\) utility of edge2 before swap
6: \( u_{1c2} \leftarrow \text{getEdgeValue(edge1.src, edge1.dest, c2)}; \) \(\triangleright\) utility of edge1 after swap
7: \( u_{2c1} \leftarrow \text{getEdgeValue(edge2.src, edge2.dest, c1)}; \) \(\triangleright\) utility of edge2 after swap
8: return \((u_{1c2} + u_{2c1}) \times T_r) - (u_{1c1} + u_{2c2})\)
9: end procedure

(i) direct neighbors, and (ii) random vertices. Direct neighbors of a vertex \( p \) are a set of vertices that are directly connected to \( p \). Every vertex has knowledge about its directly connected neighbors. Since some vertices may have a very large degree, this local search could become exhaustive for such vertices if they have to check each and every of their neighbors. Hence, vertices only consider a fixed-size random subset of their direct neighbors.

Vertices choose their partners for color exchange first from their direct neighboring. To increase the chance of finding a swap partner, vertices also consider a few random vertices from the graph. This random subset of vertices could be acquired through a peer sampling service [22, 25, 26, 127] or random walk [128] that is continuously running on all the graph vertices. In our previous work [56] we have extensively discussed these two vertex selection policies (i.e., direct and random) and how they can be realized. Moreover, we showed that the best outcome is achieved while the hybrid of direct and random neighbors are taken into account. We, therefore, use the hybrid policy, where as shown in Algorithm 15 (Line 5) and Algorithm 17 vertices first check a subset of their direct neighbors, and then if they do not succeed, they check some random vertices.

Swap Heuristics

To make a decision for color exchange we consider two heuristics: (i) dominant color and (ii) edge utility.
Dominant Color (DC).

We define the dominant color of a vertex \( p \) as the color with the maximum cardinality at \( p \). That is:

\[
c_p^* = \arg \max_c |E_p(c)|
\]

With this heuristic, a vertex \( p \) looks for a partner which (i) is not internal, and (ii) has an edge with vertex \( p \)'s dominant color. If vertex \( p \) finds such a vertex, it exchanges with that vertex one of its non-dominant colors for the dominant color. In other words, every vertex tries to unify the color of its edges, in order to reduce its energy. Since the global energy of the graph is sum of all vertices’ energy, this optimization has the potential to lead us toward a globally optimal state. Although condition (i) prevents disturbing those vertices that are already stabilized, vertices are still acting very greedily and do not consider the benefits of the other endpoint of the edge that they are negotiating over. Consequently, this policy could end up in contention between neighboring vertices, and the color of some edges might fluctuate. In Section 9.4, we will study the evolution of the partitioning under such policy.

Edge Utility (EU).

An alternative policy is to assign a utility value to every edge based on the cardinality of the colors at its endpoints. The main idea of this heuristic is to check that whether exchanging the color of two edges decreases the energy of the their connected vertices or not. If it does, two edges swap their colors, otherwise they keep them. To every edge \( e_{pq} \) (with two endpoints \( p \) and \( q \)) we assign a value \( v \), with respect to color \( c \), that indicates the relative number of neighboring edges of \( e \) with color \( c \). That is:

\[
v(e, c) = \begin{cases} 
\frac{|E_p(c)|-1}{|E_p|} + \frac{|E_q(c)|-1}{|E_q|} & \text{if } c = \pi_e \\
\frac{|E_p(c)|}{|E_p|} + \frac{|E_q(c)|}{|E_q|} & \text{otherwise}
\end{cases}
\]

Note, in the first case, \( E_p(c) \) and \( E_q(c) \) include edge \( e \), and that is why we need to decrement them by one. Next, the objective is to maximize the overall value of edges during the color exchange process. More precisely, vertex \( p \) exchanges the color of its edge \( e_{pq} \) with the color of another edge \( e'_{p'q'} \), if and only if:

\[
v(e, c') + v(e', c) > v(e, c) + v(e', c')
\]

where \( c = \pi_e \) and \( c' = \pi_e' \). Hence, the swap utility is calculated as follows:

\[
\text{utility} = (v(e, c') + v(e', c)) - (v(e, c) + v(e', c'))
\]

Simulated Annealing.

Since our swap policy is based on local optimization with limited information at each vertex and no central coordinator, it is prone to getting stuck in local optima.
9.4. EXPERIMENTS

Therefore, we need to employ some techniques to help it get out of local optima and move towards better configurations over time. To achieve this goal, we use simulated annealing [125] with constant cool down rate. Two parameters of the simulated annealing are the \textit{initial temperature} \( T_0 \) and the \textit{cool down rate} \( \delta \). The \textit{temperature} at round \( r \) is then calculated as follows:

\[
T_r = \max(1, T_0 - r \cdot \delta)
\]

Finally, as shown in Algorithm 18 we bias the utility computation with the value of temperature at each round, as follows:

\[
\text{utility} = ((v(e, c') + v(e', c)) \times T_r) - (v(e, c) + v(e', c'))
\]

9.4 Experiments

In this section, we first introduce the datasets and metrics that we used for evaluating our solution. Then, we study the impact of our simulated annealing parameters on the partitioning quality. Next, we show how different policies, introduced in Section 9.3, perform. We also measure the performance of these policies in scale, and compare them to two state of the art solutions.

To evaluate Dena we used four graphs of different nature and size for evaluating our ideas. These graphs and some of their properties are listed in Table 9.1. Note, graphs Astroph and Email-Enron have power-law degree distribution. We measure the following metrics to evaluate the quality of the partitioning:

- \textit{Vertex-cut}: this metric counts the number of times that graph vertices has to be cut. That is, a vertex with one cut has replicas in two partitions, and a vertex with two cuts is replicated over three partitions. This is an important metric, because when a graph vertices are scattered over several partitions, every computation that involves a modification to a vertex, should be propagated to all the other replicas of that vertex, for the sake of consistency. Therefore, vertex-cut directly affects the required communication cost of the partitioned graph.

- \textit{Normalized vertex-cut}: this metric calculates the vertex-cut of the final partitioning relative to the random partitioning, thus, it shows to what extent the algorithm can reduce the vertex-cut.

<table>
<thead>
<tr>
<th>Graph Name</th>
<th></th>
<th></th>
<th>power-law</th>
<th>Avg. Clustering Coeff.</th>
<th>Diameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data [135]</td>
<td>2851</td>
<td>15093</td>
<td>no</td>
<td>0.486</td>
<td>79</td>
</tr>
<tr>
<td>4elt [135]</td>
<td>15606</td>
<td>45878</td>
<td>no</td>
<td>0.408</td>
<td>102</td>
</tr>
<tr>
<td>Astroph [136]</td>
<td>17903</td>
<td>196972</td>
<td>yes</td>
<td>0.6306</td>
<td>14</td>
</tr>
<tr>
<td>Email-Enron [136]</td>
<td>36692</td>
<td>367662</td>
<td>yes</td>
<td>0.4970</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 9.1: Datasets used in the experiments
• **Standard deviation of partition sizes:** this metric measures the Standard Deviation (STD) of normalized size of the partitions. More precisely, we first measure size of the partitions relative the average (expected), i.e., in a perfect balanced partitioning the normalized size should be 1. We then, calculate how much the normalized size deviates from 1.

**Tuning the parameters**

We conducted several experiments to tune the two parameters of the simulated annealing, namely $T_0$ and $\delta$. For these experiments we selected the Data graph (Table 9.1) and $k = 2$. As shown in Figure 9.3a, the vertex-cut decreases when $T_0$ increases. However, Figure 9.3b illustrates that this improvement is achieved in a higher number of rounds, that is, a bigger $T_0$ delays the convergence time. Similarly, a smaller $\delta$ results in a better vertex-cut, at the cost of more rounds. In other words, $T_0$ and $\delta$ are parameters of a trade-off between vertex-cut and the convergence time and can be tuned based on the priorities of the applications.

Moreover, we found out that for a larger $k$, it is better to choose a smaller $\delta$, because when the number of partitions increases, the solution space expands and it is more likely for the algorithm to get stuck in local optima. Unless otherwise mentioned, in the rest of our experiments, we use $\delta = 0.0005$ for $k = 32$ and $k = 64$, and $\delta = 0.001$ for other values of $k$.

**Performance**

We observe the evolution of vertex-cut over time on different graphs, with two different swap policies: (i) $DC$, i.e., dominant color, and (ii) $EU$, i.e., edge utility. For this experiment $k = 4$, and the results are depicted in Figure 9.4. As shown, the main gain with the DC policy is acquired in the very beginning, when the vertex-cut drops sharply. After the first few iterations, although the vertex-cut does not
9.4. EXPERIMENTS

decrease considerably, it keeps on decreasing and does not converge. In contrast, the EU policy results in a lower vertex-cut and converges after the simulated annealing process stabilizes.

It is interesting to note, for the first two graphs, the DC policy can produce a better vertex-cut in a short time, but in the long run, the EU policy outperforms it. For the other graphs in Figures 9.4d and 9.4c, the EU policy is always performing better. This is due to the different structural properties of these graphs. More precisely, Astroph and Email-Enron are power-law graphs (Figures 9.4d and 9.4c), that is the degree distribution of graph vertices resembles a power-law distribution. More structural properties of these graphs are listed in Table 9.1.

We also measure the vertex-cut for various number of partitions. For this experiment, we only use the EU policy. Figure 9.5a depicts how the vertex-cut changes for various number of partitions. To better understand this result, we also report the vertex-cut of Dena relative to that of a random partitioning in Figure 9.5b. As shown, Dena reduces the vertex-cut to nearly 10-15% for Data and 4elt graphs, and to 20-30% for our power-law graphs.
Comparisons to the state of the art

In this section we show that our solution outperforms two state-of-the-art solutions in that it produces partitions with equal size, while requiring a very low vertex-cut. First, we use an edge-cut partitioner, Ja-be-Ja [56] to partition the graph. Then, the cut edges are randomly assigned to one of the partitions, where their endpoints belong to. This is similar to the example in Figure 9.2b. We also compare Dena to the state-of-the-art vertex-cut partitioner by Alessio et al. [55], which includes two policies of its own, namely D-fep, and D-fep Variant. This experiment is performed on Astroph and Email-Enron graphs with $k = 20$. To make the comparisons more meaningful, we report the normalized vertex-cut, that is the vertex-cut relative to that of a random partitioning. As shown in Figure 9.6a, Ja-be-Ja produces the minimum vertex-cut. However, Figure 9.6b shows that the partition sizes are very unbalanced. The vertex-cuts of D-fep and its variant are more than Ja-be-Ja, but their partition sizes are much more balanced. Dena has a better vertex cut than D-fep, while the partition sizes are nearly equal.

As explained in Section 9.4, the convergence time of Dena is independent of the graph size and is mainly affected by the parameters of the simulated annealing process. While this is true for Ja-be-Ja, [55] shows that both D-fep and its variant converge in only very few rounds and produce very good vertex-cuts for graphs Astroph and Email-Enron. However, as depicted in Figure 9.6b these algorithms do not maintain the balance of the partition sizes. In fact, without proper coordination, the standard deviation of the partition size distribution could grow to prohibitively large levels. Dena, however, maintains the initial distribution of edge colors, and can even be used to produce partitions of any desired size distribution, with a better vertex-cut. This comes, however, at the cost of longer running time.

9.5 Related Work

In this section we study some of the existing work on both edge-cut and vertex-cut partitioning.
9.5. RELATED WORK

Edge-cut partitioning
A significant number of algorithms exist for edge-cut partitioning [37, 51, 137–141]. These algorithms can be classified into two main categories: (i) centralized algorithms, which assume cheap random access to the entire graph, and (ii) distributed algorithms.

A common approach in the centralized edge-cut partitioning is to use Multilevel Graph Partitioning (MGP) [37]. METIS [139] is a well-known algorithm based on MGP that combines several heuristics during its coarsening, partitioning, and uncoarsening phases to improve the cut size. KAFFPA [51] is another MGP algorithm that uses local improvement algorithms based on flows and localized searches. There exist also other works that combined different meta-heuristics with MPG, e.g., Soper et al. [142] and Chardaire et al. [143] used Genetic Algorithm (GA) with MPG, and Benlic et al. [144] utilized Tabu search.

Parallelization is a technique that is used by some systems to speedup the partitioning process. For example, PARMETIS [140] is the parallel version of METIS, KAFFPAE [145] is a parallelized version of its ancestor KAFFPA [51], and [121] is a parallel graph partitioning technique based on parallel GA [122].

Although the above algorithms are fast and produce good min-cuts, they require access to the entire graph at all times, which is not feasible for large graphs. Ja-be-Ja [56] is a recent algorithm, which is fully distributed and uses local search and simulated annealing techniques [125] for graph partitioning. In this algorithm each vertex is processed independently, and only the direct neighbors of the vertex, and a small subset of random vertices in the graph need to be known locally. DIDIC [146] and CDC [147] are two other distributed algorithms for graph partitioning, which eliminate global operations for assigning vertices to partitions. However, they may produce partitions of drastically different sizes.

Vertex-cut partitioning
While there exist numerous solutions for edge-cut partitioning, very little effort has been made for vertex-cut partitioning. SBV-Cut [148] is one of the recent work
for vertex-cut partitioning. The authors proposed a solution to identify a set of balanced vertices that can be used to bisect a directed graph. The graph can then be further partitioned by a recursive application of structurally-balanced cuts to obtain a hierarchical partitioning of the graph.

PowerGraph [42] is a distributed graph processing framework that uses vertex-cuts to evenly assign edges of a graph to multiple machines, such that the number of machines spanned by each vertex is small. PowerGraph reduces the communication overhead and imposes a balanced computation load on the machines. GraphX [43] is another graph processing system on Spark [149, 150] that uses a vertex-cut partitioning to improve its performance.

ETSCH [55] is also a graph processing framework that uses a distributed vertex-cut partitioning algorithm, called DFEP [55]. DFEP works based on a market model, where the partitions are buyers of vertices with their funding. Initially, all partitions are given the same amount of funding. The algorithm, then, proceeds in rounds, such that in each round, a partition \( p \) tries to buy edges that are neighbors of the already taken edges by \( p \), and an edge will be sold to the highest offer. There exists a coordinator in the system that monitors the size of each partition and sends additional units of funding to the partitions, inversely proportional to the size of each partition.

9.6 Conclusions

We presented JA-BE-JA-VC, a distributed and parallel algorithm for vertex-cut partitioning. JA-BE-JA-VC partitions edges of a graph into a given number of clusters with any desired size distribution, while the number of vertices that have to be replicated across clusters is low. In particular, it can create balanced partitions while reducing the vertex-cut. JA-BE-JA-VC is a local search algorithm that iteratively improves upon an initial random assignment of edges to partitions. It also utilizes simulated annealing to prevent getting stuck in local optima. We compared JA-BE-JA-VC with two state-of-the-art systems, and showed that JA-BE-JA-VC not only guarantees to keep the size of the partitions balanced, but also outperforms its counterparts with respect to vertex-cut.
Chapter 10

Parallel Community Detection For Cross-Document Coreference

Fatemeh Rahimian, Sarunas Girdzijauskas, and Seif Haridi

Submitted to The 52nd Annual Meeting of the Association for Computational Linguistics (ACL 2014).
Parallel Community Detection For Cross-Document Coreference

Fatemeh Rahimian†‡, Sarunas Girdzijauskas‡ and Seif Haridi†

†Swedish Institute of Computer Science (SICS)
‡KTH - Royal Institute of Technology
{fatemeh, seif}@sics.se
sarunasg@kth.se

Abstract

This paper presents a highly parallel solution for cross-document coreference resolution, which can deal with billions of documents that exist in the current web. At the core of our solution lies a novel algorithm for community detection in large scale graphs. We operate on graphs which we construct by representing documents’ keywords as nodes and the co-location of those keywords in a document as edges. We then exploit the particular nature of such graphs where coreferent words are topologically clustered and can be efficiently discovered by our community detection algorithm. The accuracy of our technique is considerably higher than that of the state of the art, while the convergence time is by far shorter. In particular, we increase the accuracy for a baseline dataset by more than 15% compared to the best reported result so far. Moreover, we outperform the best reported result for a dataset provided for the Word Sense Induction task in SemEval 2010.

10.1 Introduction

Resolving entities in a text may not always be a difficult task for humans. When one comes across Mercury in an article about the solar system, they instantly think of Mercury, the planet, and not about Mercury, the chemical element or Freddie Mercury. For a computer though, such a disambiguation requires a considerable amount of processing. This problem, i.e., the task of disambiguating manifestations of real world entities in various records or mentions, is known as Entity Resolution or Coreference Resolution. Often disambiguation is required across multiple documents. Given a set of such documents with an ambiguous mention (Mercury, for example), the Cross-Document Coreference problem seeks to group together those documents that talk about the same entity in real world (e.g., one group for the planet, one for the chemical element, etc.).

This problem is challenging because: (i) often the number of underlying entities and their identities are not known (e.g., we do not know how many different Mercuries are to be discovered), and (ii) the number of possible classifications grows exponentially with the number of input documents.

A widely used approach to this problem, known as Mention-Pair model, is to compute a pair-wise similarity value based on the common keywords that exist
in each pair of documents [58]. If two documents are found similar more than a predefined threshold, they are classified together. Finally, a clustering step is required to partition the mentions into coreferent groups. The clustering itself is a challenging task and is known to be NP-hard. In the related work section, we discuss some of the approximate solutions that address this problem. As we will see, the high complexity of the Mention-Pair model renders it impractical for web-scale coreference, where we have to process millions of documents in a reasonable time.

(a) Fragments of several documents, all with a mention of “Mercury”, which is ambiguous. The underlined words, represent the context of the the ambiguous word in each document. Initially each document is assigned to a unique color.

(b) Each context words is represented by a node in the graph and gets one or more color(s) that correspond(s) to the document(s) it occurred in. An edge between two nodes implies the collocation of them in the same document. Node border colors indicate the dominant color in the vicinity of each node.

(c) The final coloring scheme identifies the communities of the graph. Mercury 1, 4, and 5, all belong to the red community, thus, are considered to be coreferent. Likewise, Mercury 2, 3, and 6 are considered to be coreferent because of the aqua community. The unused colors reside in a repository.

Figure 10.1: The main steps towards coreference resolution

In this paper we propose a novel approach to coreference resolution, which does not require separate classification and clustering steps. Instead, we transform the problem to a node-centric graph processing task. This enables us to take advantage of the recent advances in graph processing frameworks, such as GraphChi [59] or GraphLab [1], and apply our algorithm to extremely large graphs.

To construct the graph, we create two types of nodes. One type represents the ambiguous word, which we assume is given in advance. Another type of nodes represents the unambiguous words that surround the ambiguous word in each document. Since we do not know whether or not different mentions of the ambiguous word are referring to the same real-world entity, we create as many nodes as the number of documents mentioning them. The unambiguous words might as well appear in multiple documents. For them, however, we do not create a new node, if
they already exist. Finally, we add an edge between two nodes, if their corresponding words co-occurred in the same document. Consequently, each single document is represented by a full mesh, or clique, of all its keywords.

The constructed graph for our Mercury example is depicted in Figure 10.1b. As shown, some cliques overlap, which indicates that their corresponding documents have a similar context. In fact, the main insight to our work is that the topological community structure of the constructed graph identifies similar contexts and thus, is an accurate indicator of the coreference classification. Based on this fact, we propose a novel community detection algorithm for coreference resolution. Our algorithm is diffusion based and exploits the fundamentals of flow networks. In such a network each node has a capacity and each edge can transfer a flow, just like a pipe, between two nodes. We envision multiple flows in our graph, one per community. To distinguish these flows, we assign a distinct color to each of them.

Initially each single document constitutes a distinct community, i.e., it will be assigned to a unique color. All the nodes that belong to a document will get a unit of the color of their document. Therefore, those nodes that are shared between documents, will receive multiple units of colors. However, each node always identifies itself with only a single color, which has the highest collective volume in its neighborhood, so-called the dominant color. The initial coloring scheme of our Mercury graph is shown in Figure 10.1b. Nodes continuously exchange parts of their colors with their neighbors by diffusing the colors through their links. Therefore, the available volume of color at nodes, and accordingly the dominant color in their vicinity, changes during the course of algorithm. We will show that with appropriate diffusion policies it is possible to accumulate one distinct color in each of the well connected regions of the graph, e.g., as in Figure 10.1c. Finally, the ambiguous nodes that end up having the same dominant color are considered to be coreferent.

Since our constructed graph is sparse, the overhead of such computation remains low (for complexity analysis see Section 10.3). Moreover, we can produce more accurate results, compared to the state of the art. This twofold gain is owed to the combination of two ideas, that constitute our main contributions:

- a technique for transforming the expensive coreference problem, into a graph problem, in which the coreferent words belong to the same topological community structure. The graph that we construct is sparse, because those documents that have dissimilar contexts, will have very few or even no direct connections. The computations on the graph are performed per edge basis, i.e., only if there is an edge between two nodes, they will communicate some flows. Hence, the irrelevant documents which are weakly connected, if not disconnected, will not impose any computation in the graph. At the same time, a more thorough search of the solution space is possible, as we are not limited to pair-wise similarity discoveries only. Instead, similarity between any number of documents is naturally captured within the community structures that emerge from the inter-linked context words.
• a novel node-centric diffusion-based community detection algorithm that mainly uses local knowledge of the graph at each node. Hence, it allows for highly parallel computations and usage of the existing graph processing frameworks.

We run our algorithm on different datasets, which are transformed to graphs with distinct structural properties. For example, on a baseline dataset for person name disambiguation, we produce a classification with an F-score 15% higher than that of the state of the art algorithm by Singh et al. [60]. Moreover, on a dataset provided in the Word Sense Induction task of SemEval 2010, we achieved as good F-score as the best reported result. However, we considerably outperform the other solutions with respect to a complementary accuracy metric, which measures the average number of items in each clusters.

10.2 Terminology

The main terms that we are going to use hereafter are:

• \textit{Entity} is a unique representation of someone/something in the real world, e.g., “Paris Hilton”.

• \textit{Mention} (or simply \textit{word}) is a literal manifestation of a real world entity. Since multiple entities could share a similar name, mentions can sometimes be ambiguous, e.g., “Paris” in the sentence “Paris is nice.”, which could refer to the capital city of France, the city in Texas, Paris Hilton, or many other possibilities.

• \textit{Context} of an ambiguous mention $M$ is a set of unambiguous mentions surrounding mention $M$. For example, in Figure 10.1a, the underlined words in each sentence constitute the context of the ambiguous word Mercury in that sentence. Note, extracting an appropriate context for a mention is not trivial, and is an active field of research. However, this task is out of the scope of this paper, and we only use the extracted context as an input to our algorithm.

• \textit{Community} or \textit{cluster} in a graph is a densely connected component. \textit{Community detection} is the task of grouping the vertices of the graph into clusters taking into consideration the edge structure of the graph in such a way that there should be many edges within each cluster and relatively few between the clusters. Note, this problem is different from \textit{graph partitioning}, where the goal is to divide the graph into a predefined number of roughly equal size components, such that the number of edges crossing different component is minimum (a.k.a., the \textit{min-cut} problem). In Community detection we neither know the number of existing communities nor the size of them.
10.3 Solution

Given an ambiguous word \(^1\) and a set of documents with preprocessed context words, we classify the ambiguous words into coreferent groups in two main steps: (i) Graph Construction, and (ii) Community Detection.

Graph Construction

We create a graph using the ambiguous and unambiguous words as nodes. For the ambiguous word we always create a new node per document. We call such nodes the target nodes, as those are the nodes we aim to classify. For the non-ambiguous words, if a matching node already exists, we reuse it; otherwise, we create a new node. Moreover, if two words have appeared together in a document, we add an edge between the nodes representing them. The resulting graph is weighted, as some words may frequently appear together in multiple documents. More precisely, the weight of each edge is proportional to the number of documents that contain both endpoint nodes of that edge.

The graph corresponding to our Mercury example is depicted in Figure 10.1b. Each mention of Mercury has a distinct node in the graph, tagged with its sentence identifier. Edge thickness (weight) between pair of nodes is relative to the number of times the two words have appeared together across different sentences. Now our task is to classify these Mercury nodes into several (ideally two, in this case) clusters, where all the Mercuries in a cluster refer to the same entity. Note, we do not know, in advance, how many different Mercuries are expected to be resolved, i.e., we do not know the number of expected clusters/communities.

Community Detection

We propose a massively parallel diffusion based algorithm for community detection. Without lack of generality, we assume that the algorithm proceeds in rounds.

Initialization

Initially, each documents is associated with a distinct color. Every node is given a unit of color corresponding to the color of the document that holds it. Accordingly, if a node belongs to multiple documents, it receives multiple colors.

Each node identifies itself with a dominant color, which is the color with the highest total volume among the node and its neighbors \(^2\). The dominant color of a node also indicates the community that the node belongs to.

\(^1\)Our solution is not limited to a single ambiguous word and we can have multiple of such words. However, for the sake of clarity, we consider the case with one ambiguous word only. To discover which word(s) is(are) ambiguous, is a different problem which is orthogonal to our work and out of the scope of this paper.

\(^2\)We break the ties with some globally known ordering of colors, e.g., least color id.
Diffusion

Every node repeatedly runs the diffusion algorithm, until the convergence criteria, defined in section 10.3, is satisfied. In each round a node sends out some amount of color to its neighboring nodes, and likewise, receives some amount of color from its neighbors. The key point is to decide which color or colors should be sent out and in what quantity. This decision is made locally at each node and is based on one main objective, that is, each node tries to change its color to the one that is dominant in its neighborhood, i.e., the color with the largest collective quantity across its neighbors and the node itself.

The effort that a node makes to change its color to the dominant color (or maintain it, if it already has the dominant color) consists of two main forces: (i) an attraction force that conserves the dominant color, and (ii) a repulsion force that evacuates all the non-dominant colors. The algorithm is further completed with a recycling mechanism that nourishes the diffusion by collecting the colors from the regions where they are non-dominant and putting them back into the regions where they can become influential again.

Note, since the colors only flow through the edges of the graph to the neighboring nodes, disjoint clusters will never get the same color, as there will be no link connecting them and carrying the flows. This property of the solution is desirable, because disjoint clusters indicate disparate contexts and are not expected to be in the same coreference chain.

The diffusion algorithm at each node is composed of the following three rules:

- **Attract:** Keep fraction $\alpha$ of the dominant color, and divide the rest equally between neighbors.

  The attraction force should be applied with a subtlety. If nodes are too greedy, meaning that they do not let any dominant color to leak out, then there will be no chance that their neighboring nodes, which have a different color, will get influenced and change their color. On the other hand, if the leakage is too high, all the colors will freely and rapidly explore the entire graph, and the concentration of a distinct dominant color in each of the community structures will not take place. However, if nodes allow for an appropriate amount of leakage, i.e., $\alpha$, over time they not only maintain their color, but also might be able to expand their territory and let more nodes into their community. Parameter $\alpha$ determines to what extent the communities are likely to merge, thus, controls the resolution of the detected communities. A bigger $\alpha$ produces more communities with smaller sizes, whereas a smaller $\alpha$ is likely to produce fewer communities of a bigger size.

- **Repel:** Divide all the non-dominant colors equally between the neighbors.

  With this rule, nodes evacuate the non-dominant colors. If a color is not dominant in any region, it will be subject to this repulsion force all over the graph, and thus it will be highly fragmented. Consequently, no node will
identify itself with that color, as though the color has disappeared from the graph. On the other hand, there are colors that are recessive in some region, but dominant in another region. This repulsion force allows such colors to flow in the graph and be absorbed in the regions, where there is an attraction force for them.

- **Recycle:** If surrounded by neighbors of the same dominant color, send all the non-dominant colors (if any) to a repository, and get some dominant color from the repository (if there is any).

The repulsion force works blindly, meaning that the non-dominant colors are sent to all directions, with the hope that they might encounter an attraction force in the neighboring nodes. But a node may be completely surrounded by nodes of the same dominant color as the node itself. We call such a node, an *interior node*. Consider a node with the dominant color *red*, surrounded by neighbors of the same dominant color. If such a node receives some *blue* color, sending it to the neighbors, would only disturb the *red* territory. Instead of blindly repelling non-dominant flows, interior nodes take a more efficient approach, by sending the non-dominant colors directly to a repository, where all such colors are accumulated. The repository is accessible by all the nodes and acts as a container for the colors that are collected from the graph. It also keeps track of the number of interior nodes per color.

The recycling process is completed by putting the abandoned colors back in the regions where they are dominant. When an interior node send some non-dominant color to the repository, in return, the repository send a share of the the node’s dominant color back to it. The amount of this share depends on the available amount of the dominant color in the repository as well as the number of interior nodes that require it. Since the repository knows the number of such nodes, it divides the color equally between them. The augmented amount of the dominant color in the interior nodes will then help taking over the neighboring regions, if they are strongly connected to the reinforced community. Those colors that are not dominant in any region of the graph will remain in the repository for ever, and will never again flow in the graph. Consequently, over time, nodes of the graph have to deal with fewer and fewer number of colors, which makes their computations much faster and more efficient.

**Convergence**

When the coloring scheme of the graph does not change any more, and we will remain a a stationary state, we consider the algorithm is converged. At the convergence time, the coloring scheme of the whole graph determines the clustering of the nodes. More precisely, the target nodes that have the same dominant color at the end, will be considered as coreferent. Also, the number of different colors for the target nodes indicate the number of different references for the ambiguous word.
Complexity

In this section, we give some bounds on the complexity of our algorithm. First of all, if $N$ is the number of documents, and if we choose on average $c$ words from each document, the number of nodes would be at most $c \times N$, which is $O(N)$. In each round, every node communicates with its neighboring nodes. If $d$ indicates the degree of a node, then the communication complexity of the algorithm in each round is $O(N \times d)$, where $d$ represents the average node degree. Hence, the overall complexity will be $O(N \times d \times \text{rounds})$.

The maximum number of rounds before convergence is proportional to the time required for the existing colors in the graph to spread out to all the reachable regions. Therefore, it is proportional to the diameter of the biggest connected component in the constructed graph. It is important to note that the diameter of the graph is inversely proportional to the average node degree, means if the average node degree is higher, the expected number of rounds is lower. Also note that our constructed graph is sparse, i.e., the average node degree is far less than $N$. Hence, in practice, $O(N \times d \times \text{rounds})$ becomes significantly smaller than $O(N^2)$, which makes the large-scale coreference feasible.

10.4 Experiments

Our algorithm is vertex-centric and can be deployed over any of the existing graph processing frameworks, such as Graphlab [1] or Graphchi [59], which are proven to be scalable, efficient and fast. At the moment we are using Graphchi framework [59], which is a disk-based system for efficient computations on graphs with billions of edges. The repository for the colors, is implemented as a shared object among the nodes. Graphchi, as well as other frameworks, provides means of working with objects that can be accessed by all the nodes concurrently. We ran several experiments to tune parameter $\alpha$ that is used in the Attract rule. We concluded that the best result is achieved when $\alpha$ is set to $\frac{2}{3}$.

Metrics

We compare our detected communities (of target words) with the true classification of coreferent words, using a metric, called $B^3$, which breaks down to a few other metrics. For every target word (document):

- $B^3\text{Precision}$ is the fraction of detected coreferents that are actually coreferent with it.
- $B^3\text{Recall}$ is the fraction of its actual coreferents that are detected as being coreferent with it.

We then calculate the overall $\text{Precision}$ and $\text{Recall}$ by taking the average of $B^3\text{Precisions}$ and $B^3\text{Recalls}$ for all the target words (documents). To clarify these
metrics, note that if we put each document in a separate community, we will have 100 percent precision, because no irrelevant words are wrongly grouped together. This naive way of clustering shows that precision alone, can not give us an idea how accurate our clustering is. Hence, we need other complementary metrics. Now, if we put all the documents in one single community, we will have 100 percent recall. This is again an extreme case, where we do not produce meaningful clusters, thus, the precision will be very low. Therefore, we use the harmonic mean of these two metrics, which is widely known as $F_1$-Score. Since we compute the precision and recall based on $B^3$, our $F_1$-score is also a $B^3$ score:

$$B^3F_1 = 2 \times \frac{B^3Precision \times B^3Recall}{B^3Precision + B^3Recall}$$

Hereafter, whenever we use the terms precision, recall or $F_1$-score we always refer to the $B^3$ variants of these metrics. Moreover, to be comparable to other reported results for the SemEval dataset, we use the evaluation script, given for the same task. In addition to precision, recall and $F_1$-score, this script measures the average number of items in each cluster. Therefore, we also measure this additional metric in our SemEval experiments.

**Results**

We have used three different datasets, namely Chris Andreson, John Smith, and SemEval. For the first two datasets we have used OpenCalais [151] to extract the name entity mentions as context. For the SemEval task we have used all the context words. The results for each dataset are reported separately.

**Chris Anderson Corpus**

This corpus, provided by our industrial partner X, is extracted from 1185 documents from the web. Each document contains a reference to one out of 8 persons, named Chris Anderson.

The graph produced for this dataset is depicted in Figure 10.2. We monitored the progress of our community detection algorithm and have reported the accuracy metrics in each round in Figure 10.3. As shown, in the very beginning the precision is 100%, because each document is considered to be a distinct community, thus, no two documents are wrongly classified together. However, the recall is very low, because no two documents are correctly classified together either. Over time, as the communities merge, recall improves considerably, at the cost of a small downgrade in precision. Overall, the $F_1$-score increases significantly up to a certain level, by then the community detection algorithm has converged. We observe that after only 7 rounds the community detection algorithm has converged and the final communities are detected with more than 85% accuracy.

Note, the fast convergence of the algorithm is due to the topological properties of this specific graph. As shown in Figure 10.2, the small non-ambiguous nodes
constitute only a small fraction of the nodes in the graph. This means, the ambiguous words that are coreferent has extremely similar context. Thus, their cliques are tightly connected, and it takes very few steps for one dominant color to take over the tightly connected regions of the graph. Moreover, some of the communities that have the same color in Figure 10.2, are not even connected. This means that the vector of the terms extracted from their corresponding document are not informative enough for the two documents to be classified together. In such cases, we do not expect our algorithm to merge the two communities, as there is no way any color will flow from one to another. In fact, no other solution could do any better with only this incomplete information.

John Smith Corpus

This corpus, originally introduced by Bagga and Baldwin [58], contains 197 New York Times articles about 35 different people named John Smith. Each article mentions a single John Smith. Twenty four clusters contain a single document, while the rest contain the following numbers of documents: 2, 2, 4, 4, 5, 9, 15,
20, 22, 88. This dataset gives us a different graph type, as shown in figure 10.4.

The progress of precision, recall and $F_1$-score over time is depicted in figure 10.5. Again we start with 100% precision and a very low recall. As shown, it takes longer for the algorithm to converge. This is expected, because as opposed to Chris Anderson graph, there are so many small clusters connected by very few links (See Figure 10.4). Here, the fraction of non-ambiguous (small dots) to ambiguous nodes (depicted with a larger size) is not negligible. Therefore, it takes longer for a color to reach out to other regions of the graph and explore any potential community merge possibilities. Finally, when we do not observe any more change, the $F_1$-score is over 80%. It is important to compare this result with other existing solutions that have worked with this same dataset. For example, in a nice work by Singh et al. [60] from Google, which has also proved scalable, $F_1$-score is reported to be 66.4%. Also, the best reported result so far is 69.7%, by Rao et al. [152]. This means we have increased the accuracy more than 15% compared to the state-of-the-art.

**SemEval 2010, WSI task**

This dataset [153] was given as the test data for the Word Sense Induction and Disambiguation task in SemEval 2010. As opposed to the John Smith and Chris Anderson datasets, the work on this dataset is not limited to person name entity resolution. The texts come from various news sources including the Wall Street Journal, CNN, ABC and others. It includes 50 ambiguous nouns and 50 ambiguous verbs. Each test instance consisted of a maximum of three sentences.

Here again we construct our graph for each word separately and apply our community detection algorithm. The evaluation script provided in the task, enables us to report the results for nouns only, verbs only, or all the words. The results are reported in Figure 10.6. Our solution produces clusterings with $F_1$-scores 62.2, 70.7, and 56.4 for “all”, “verbs”, and “nouns”, respectively. Among the reported results so far, only *Duluth-WSI-SVD-Gap* [153] has a slightly higher $F_1$-Score than us.
(63.3, 72.4, and 57.0 percent accuracy for “all”, “verbs” and “nouns”, respectively). However, the average number of clusters found by this solution (\(\#Cl\) in Figure 10.6) is 1.02, which is far from the actual numbers (4.46 for nouns, 3.12 for verbs, and 3.79 for all). In fact, this result is very close to a naive clustering, known as the most frequent clustering (MFS), in which all the mentions of a word are classified into one single cluster, hence the average number of clusters per word is 1. With our solution, however, the average number of clusters is 3.61, which is much closer to the actual number of clusters in the golden truth. In [154], which bears the closest resemblance to our work, the best reported F1-score is 63.4.

10.5 Related Work

We organize this section in two main parts. In the first part we summarize the related work on Coreference, which is the problem that we are addressing. Since our solution is a community detection algorithm, in the second part we survey the existing solutions for graph clustering and community detection.

Related Work in Cross-Document Coreference

Coreference is one of the complicated NLP problems that has received a lot of attention in the community, and has yielded numerous solutions. Several publications has already surveyed the coreference research in details [155–157]. Due to the space limitations, we focus on the related work in cross-document coreference resolution,
10.5. RELATED WORK

which is the problem addressed in this paper. Since we use a graph-based solution, we also overview some of the existing graph-based approaches.

One of the most well-known approaches to cross-document coreference is presented by Bagga and Baldwin [58]. They used the original vector space model introduced in [158], in which every document is represented by a vector of its terms. These terms are weighted proportional to their frequency in the text. Then a similarity measure between each pair of documents is computed based on the common terms in the two documents, also considering their weights. If the similarity of two documents is above a predefined threshold, then the entity of interest in those two documents are considered to be coreferent. This approach is still widely used and several improvements to it are proposed [159–161]. These solutions generally fall into the category of mention-pair models [157] and require a final clustering step after the initial pair-wise comparisons, which is not always straightforward. This is mainly because the transitivity of coreference can not be always enforced. For example, if the similarity comparisons suggest that $A$ and $B$ are similar, as well as $A$ and $C$, but $B$ and $C$ cannot be the same, due to gender or size contradictions, then the final clustering becomes challenging. Although multiple approximate solutions to this problem are proposed [162], Ng [157] in his survey paper argues that the mention-pair model remains fundamentally weak. Various alternative approaches have thus been proposed to improve on the classical mention-pair model [163–165].

There have also been a few graph-based approaches to coreference resolution. These approaches do not require separate classification and clustering steps. Instead, they construct a graph in which nodes represent the mentions, and links represent the relation between pairs of mentions. Several interesting ideas have been proposed for how to compute the weight on the links and how to put it in use. In [166–168], for example, there can be multiple links (of different types) between nodes, thus, a hyper graph is constructed. Weights on the links are then learned through a training phase. Then the hypergraph is partitioned into multiple sub-hypergraphs by means of a spectral graph clustering or greedy partitioning. Finally, all the mentioned of a partitioned component are considered coreferent. In [169], the betweenness centrality of the nodes that may be coreferent is computed. Similarly, [170] investigated graph attributes and links to rank the similar nodes of the graph. Another interesting graph-based approach is recently proposed in [171]. First, they transform an unweighted undirected cyclic entity graph, into an unweighted, directed, acyclic one. Then, they find the maximal quasi-strongly connected components of the transformed graph. Finally, all the mentions that belong to the same component are considered to be coreferent. The main problem with these solutions is that they are computationally very expensive, thus, it is impractical to use them in web scale.

An existing work that can be applied on very large datasets is [60], which transforms the problem to a Markov chain Monte carlo (MCMC) based inference. Mentions and entities are random variables. Each mention takes an entity as its value, and each entity takes a set of mentions as its value. To scale up the MCMC-based inference, initially the entities are distributed among multiple machines. Then, in-
dependent MCMC chains are computed on each machine using only the local merge proposals. After a certain number of rounds, the entities are redistributed among machines to enable merge proposals for entities that were previously on different machines. Another scalable solution, proposed in [152], is a streaming algorithm for coreference resolution, which can work on very large datasets. The closest work to ours is perhaps by Jurgen et al. [154], which also applies a community detection algorithm on a collocation graph. As opposed to our node-centric solution, [154] uses an agglomerative algorithm for community detection, which could become expensive for big graphs.

Related work in Graph Clustering

Graph clustering problem is to divide nodes of a graph into multiple components, such that the number of edges that cross different components is minimum. This is also known as the min-cut problem, because it identifies the minimal set of edges that can be cut in order to split the graph. Two main graph clustering problems are known: (a) balanced graph partitioning, and (b) community detection. For balanced graph partitioning the number of desired components is given in advance, and there is a constraint that the components should hold roughly equal number of nodes [56, 116, 172]. However, in the community detection problem neither the size nor the number of components are known. Instead the task is to cluster the nodes into groups or communities, which are tightly connected together, but very sparsely connected to the rest of the nodes. Despite their differences, the two problems are not conceptually different, as they both strive to find the min-cut of the graph. Thus, many ideas and techniques can be applied to both of them. Here, we give an overview of the main category of graph clustering solutions. For more detailed information please refer to the existing surveys by Fortunato et al. [173] and Schaeffer et al. [174].

Spectral Clustering

Spectral clustering consists of a transformation of the initial set of nodes/edges into a set of points in the space, whose coordinates are the element of the eigenvectors. The set of points, can then be clustered by using well-known techniques, such as K-means clustering [175]. The very nice property of spectral graph theory is that we can acquire a lot of topological knowledge about the graph just by looking at its eigenvectors. For example, we can discover if the graph is connected, or even how many connected components exist in a graph. Similarly, we can find the number of existing communities in a graph. Nevertheless, the problem is computing the eigenvectors of a large graph is very costly if not impossible. We, therefore, has to look for alternative approaches to clustering, in order to process very large graphs.
Hierarchical Clustering

Hierarchical clustering approaches also aim at dividing the graph into tightly connected groups of node, where often the number and size of the groups are unknown. These techniques, therefore, allow for clustering with different resolutions, i.e., with a low resolution clustering the graph is partitioned into few big components, while we can zoom into these coarse components and find components of a finer resolution. As a result, a hierarchy of the clusters will be computed. These solutions can be classified into two main categories: agglomerative algorithms, as in [176], and divisive algorithms, as in [177]. In the first set of algorithms, we start by singles nodes in each cluster, and merge clusters that have many neighbors in common, whereas in the second set, we take the opposite approach. We start from one cluster that holds all the nodes, and then iteratively remove the edges to break the clusters into smaller pieces. Hierarchical algorithms are nice for they enable us to acquire clusters of any desired resolution. However, not all the graphs have a hierarchical topology, in which case zooming in and out of a cluster does not necessarily identifies meaningful clusters. Also, these algorithms are computationally expensive, thus, are impractical for big graphs.

Diffusion Based Clustering

The well-known theorem of Ford and Fulkerson [178] states that the min-cut between any two vertices $s$ and $t$ of a graph, i.e., any minimal set of edges whose deletion would topologically separate $s$ from $t$, carries the maximum flow that can be transformed from $s$ to $t$ across the graph. The duality of the min-cut/max-flow problem has indeed inspired a class of diffusion based algorithm, including our work in this paper, for graph partitioning and community detection. There are already several solutions to detect max-flow in graphs, including [179], [48], and DiDic [146]. DiDic uses a statistical model to diffuse flows of different colors in a graph, while it biases the flows towards well-shaped regions. The problem is we need to give it the maximum number of communities in advance, and if this number is unnecessarily large, it can slow down the process significantly. Also, since the initialization is random, each run of the algorithm on the same graph might result in a drastically different result. In effect, the final result is more affected by the initialization scheme rather than the graph topology.

10.6 Conclusion and Future Work

In this paper, we introduced a graph-based approach to coreference resolution. We showed that by using a graph representation of the documents and their context, and applying a community detection algorithm we can speed up the task of coreference resolution by a very large degree. More precisely, the complexity of our algorithm is $O(N \times \bar{d} \times \text{rounds})$, where $\bar{d}$ is the average node degree and rounds is the number of rounds before convergence. Moreover, the convergence time of
the algorithm highly depends on the topology of the constructed graph and is proportional to the diameter of the largest connected component. The accuracy of coreference resolution could also be improved at the same time, because we are able to search beyond only pair-wise comparisons. The graph that we construct enables us to discover any existing closeness/similarity between any subset of documents. Thus, we can explore the solution space more freely and more smartly.
Bibliography


