# A Generalized Algorithm for Publish/Subscribe Overlay Design and its Fast Implementation

Chen Chen[1], Roman Vitenberg[2], and Hans-Arno Jacobsen[1]

[1] Department of Electrical and Computer Engineering, University of Toronto, Canada
[2] Department of Informatics, University of Oslo, Norway
{chenchen,jacobsen}@eecg.toronto.edu, romanvi@ifi.uio.no

**Abstract.** It is a challenging and fundamental problem to construct the underlying overlay network to support efficient and scalable information distribution in topic-based publish/subscribe systems. Existing overlay design algorithms aim to minimize the node fan-out while building topic-connected overlays, in which all nodes interested in the same topic are organized in a directly connected dissemination sub-overlay. However, most state-of-the-art algorithms suffer from high computational complexity, such as $O(|V|^4|T|)$, where $V$ is the node set and $T$ is the topic set.

We devise a general indexing data structure that provides a significantly faster implementation, with $O(|V|^2|T|)$ running time, for different state-of-the-art algorithms. The generality of the indexing data structure is due to the fact that it enables edge lookup by both node degree and *edge contribution*, a central metric in all existing algorithms. When tested on typical pub/sub workloads, the speedup observed was by a factor of over 1 000, thereby rendering the algorithms more suitable for practical use. For example, under a typically Zipf distributed pub/sub workload, with 1 000 nodes and 100 topics, our new implementation completes in 3.823 seconds, while the previous alternative takes over 555 minutes.

## 1 Introduction

Publish/subscribe (pub/sub) systems constitute an attractive choice as communication paradigm and messaging substrate for building large-scale distributed systems. Many real-world applications are using pub/sub for message dissemination, such as application integration across data centers [27], financial data dissemination [3], RSS feed aggregation, filtering, and distribution [23,26], and business process management [21]. Google's GooPS [27] and Yahoo's YMB [15] constitute the distributed messaging substrates for online applications operating worldwide, TIBCO RV [3] has been used extensively for NASDAQ quote dissemination and order processing, and GDSN (Global Data Synchronization Network) [1] is a global pub/sub network enabling suppliers and retailers to exchange timely and accurate supply chain data.

In a distributed pub/sub system, so called pub/sub brokers, often connected in a federated manner as an application-level overlay network, efficiently route publication messages from data sources to sinks. The overlay of a pub/sub system directly impacts the system's performance and the message routing cost. Constructing a high-quality broker overlay is a key challenge and fundamental problem for distributed pub/sub systems that has received attention both in industry [27,15] and academia [13,24,25,18,7,10].

The notion of topic-connectivity is defined for topic-based pub/sub overlays [13], which informally speaking means that all nodes (i.e., pub/sub brokers) interested in the same topic are organized in a connected dissemination sub-overlay. This property ensures that nodes not interested in a topic would never need to contribute to disseminating information on that topic. Publication routing atop such overlays saves bandwidth and computational resources otherwise wasted on forwarding messages of no interest to the node. It also results in smaller routing tables. From a security perspective, topic-connectivity is desirable when messages are to be shared across a network among a set of trusted users without leaving this set.

Apart from topic-connectivity, it is imperative for an overlay network to have a low node degree. It costs a lot of resources to maintain adjacent links for a high-degree node (i.e., monitor the links and the neighbors [13,25]). Besides, for a typical pub/sub system, each link would have to accommodate a number of protocols, service components, message queues and so on. While overlay designs for different applications might be principally different, they all share the strive for maintaining bounded node degrees, whether in DHTs [22], wireless networks [16], or for survivable network design [19].

Several centralized algorithms have been proposed for constructing topic-connected overlays with the average node degree or the maximum node degree provably close to the optimal ones [13,24,25,10,12]. These state-of-the-art algorithms target overlay construction in a managed large cluster of up to thousands of servers where full mesh solutions exhibit scalability problems [27,15]. Such clusters are characterized by a large degree of control and relatively low churn rates (in the order of one change every hour, depending on the size of the cluster [2]), which makes centralized overlay construction a viable solution. Besides, these algorithms serve as stepping stones and comparison baselines for dynamic environments and decentralized overlay construction protocols.

However, the algorithms in [24,25,12] have the prohibitively expensive runtime cost of $O(|V|^4|T|)$ where $|V|$ is the number of nodes and $|T|$ is the number of topics. This fundamental drawback makes the algorithms non-suitable for the managed cluster environment because it takes tens of minutes or hours to compute an overlay for a realistic scale on a high-end machine. The runtime cost also limits the applicability of the algorithms as a comparison baseline.

The main contribution of this paper is that we generalize the above algorithms and come up with a new indexing data structure that supports a significantly faster implementation, with $O(|V|^2|T|)$ time efficiency. Specifically, all algorithms follow the same pattern: they iteratively add edges until the resulting overlay satisfies topic-connectivity. The data structure that we propose exhibits the following properties: (a) its initialization complexity is $O(|V|^2|T|)$, (b) the cumulative complexity of selecting an edge at all iterations is $O(|V|^2|T|)$, and (c) the amortized complexity of updating the data structure over all iterations is also $O(|V|^2|T|)$. The generality of the indexing data structure is due to the fact that it allows edge lookup by both node degree and the *edge contribution*, a central metric in the above algorithms.

To complement the theoretical analysis, we conduct comprehensive experiments under a variety of characteristic pub/sub workloads. Our experiments show that on average, for a typical pub/sub scale and interest distribution, our generalized algorithm with its efficient implementation builds the same overlay as previously known state-of-

the-art algorithms in less than $0.37\%$ of the running time. For example, under the Zipf distributed pub/sub workload, with $1000$ nodes and $100$ topics, our new implementation completes in $3.823$ seconds, while the previous alternative takes over $555$ minutes.

## 2 Related Work

The research in distributed pub/sub systems has been considering two main directions: (1) the design of routing protocols with emphasis on the efficiency and scalability of message dissemination from numerous publishers to a large number of subscribers (see for example: [28,8,20,4]) and (2) the construction of the underlying overlay topology such that network traffic is minimized (see for example: [13,24,18,7,10,25,17,12]). This paper focuses on the latter direction.

Topic-connectivity is a required property in [6,14]. It is also an implicit requirement in [8,5,7,9,17], which all aim to reduce the number of unnecessary intermediate overlay hops for message delivery using a variety of techniques.

Chockler *et al.* [13] introduced the parametrized family of *Scalable Overlay Construction (SOC)* design problems for pub/sub that captures the trade-off between the overlay scalability and the cost of message dissemination. They specifically focus on the MinAvg-TCO problem of minimizing the average node degree of the topic-connected overlay [13]. They proved the NP-Completeness of MinAvg-TCO and proposed the GM (Greedy Merge) algorithm that achieves a logarithmic approximation ratio with regard to average node degree [13]. Chen *et al.* [10] use GM as a building block for designing a divide-and-conquer approach to overlay design for pub/sub systems. This approach significantly reduces the time and space complexity of constructing a topic-connected overlay with a low average node degree.

Onus and Richa [24] analyzed the MinMax-TCO problem of minimizing the maximum degree of a topic-connected overlay network. They present the MinMax-ODA (Minimum Maximum Degree Overlay Design Algorithm) that attains a logarithmic approximation ratio on the maximum node degree. Chen *et al.* [12] focus on providing an efficient solution for MinMax-TCO by combining greedy and divide-and-conquer algorithm design techniques.

The GM and MinMax-ODA algorithms each focus on minimizing one single node degree metric, either average or maximum node degree. Each algorithm was shown to perform poorly with respect to the complementary metric. Onus and Richa [25] introduced the Low-TCO problem for minimizing both average and maximum node degrees in a topic-connected pub/sub overlay design at the same time. The authors designed the Low-ODA (Low Degree Overlay Design Algorithm), which achieves sub-linear approximations for both metrics [25].

Both MinMax-ODA and Low-ODA have the high time complexity of $O(|V|^4|T|)$, where $|V|$ is the number of nodes and $|T|$ is the number of topics. In this paper, we provide a generalization of the GM, MinMax-ODA, and Low-ODA algorithms and propose a fast implementation for the generalized algorithm with running time $O(|V|^2|T|)$. This speedup technique is also applicable to the algorithms proposed by Chen *et al.* [10,12] that can use the generalized algorithms as building blocks.

## 3 Background

In this section we present some definitions and background information essential for the understanding of the algorithms developed in this paper.

Let $V$ be the set of nodes and $T$ be the set of topics. The interest function $Int$ is defined as $Int : V \times T \rightarrow \{true, false\}$. Since the domain of the interest function is a Cartesian product, we also refer to this function as an interest matrix. Given an interest function $Int$, we say that a node $v$ is interested in some topic $t$ if and only if $Int(v,t) = true$. We then also say that node $v$ subscribes to topic $t$.

An overlay network $G(V,E)$ is an undirected graph over the node set $V$ with the edge set $E \subseteq V \times V$. Given an overlay network $G(V,E)$, an interest function $Int$, and a topic $t \in T$, we say that a sub-graph $G_t(V_t,E_t)$ of $G$ is *induced* by $t$ if $V_t = \{v \in V | Int(v,t)\}$ and $E_t = \{(v,w) \in E | v \in V_t \wedge w \in V_t\}$. An overlay $G$ is called *topic-connected* if for each topic $t \in T$, the sub-graph $G_t$ of $G$ induced by $t$ contains at most one *topic-connected component* (*TC-component*). A *topic-connected overlay* (TCO) is denoted as $TCO(V,T,Int,E)$, $TCO$ in short.

The concept of TCO is applicable to both P2P solutions for pub/sub in which the clients form the TCO and broker-based solutions in which the brokers form the TCO. It does not differentiate between publishers and subscribers. This abstraction simplifies the presentation for a theoretical and algorithmic treatment of the problem, while fully preserving its practical character. Aiming to achieve topic-connectivity while optimizing node degrees has resulted in the formulation of various problems: MinAvg-TCO for average degree [13], MinMax-TCO for maximum degree [24], and Low-TCO for both average degree and maximum degree simultaneously [25].

*Problem 1.* MinAvg-TCO$(V,T,Int)$: Given a set of nodes $V$, a set of topics $T$, and the interest function $Int$, construct a topic-connected overlay which has the least possible total number of edges (i.e., the least possible average node degree).

*Problem 2.* MinMax-TCO$(V,T,Int)$: Given a set of nodes $V$, a set of topics $T$, and the interest function $Int$, construct a topic-connected overlay with the smallest possible maximum node degree.

*Problem 3.* Low-TCO$(V,T,Int)$: Given a set of nodes $V$, a set of topics $T$, and the interest function $Int$, construct a topic-connected overlay with both low average and low maximum node degree.

MinAvg-TCO and MinMax-TCO are proven NP-Complete [13,24]. Low-TCO integrates the optimization objectives of the former problems. Approximation algorithms are proposed for these TCO construction problems, all following a greedy heuristic: the GM algorithm for MinAvg-TCO [13], the MinMax-ODA algorithm for MinMax-TCO [24], and the Low-ODA algorithm for Low-TCO [25].

## 4 Generalized Overlay Design Algorithm

In this section, we introduce Gen-ODA (Generalized Overlay Design Algorithm) as specified in Alg. 1. It captures the similarities embedded in the GM, MinMax-ODA,

and Low-ODA algorithms and offers an easy-to-specialize pattern for studying families of algorithms for solving TCO design problems. We illustrate some of the specializations of this pattern in this paper.

Gen-ODA starts with the overlay $G(V, E_{new})$ where $E_{new} = \emptyset$ so that there are $\left|\{v : Int(v,t)\}\right|$ singleton *TC-components* for each topic $t \in T$, i.e., there are $\sum_{t \in T} \left|\{v{:}Int(v,t)\}\right|$ separate *TC-components* in total. The algorithm progresses by adding edges to $E_{new}$, thus merging *TC-components* until $G(V, E_{new})$ contains at most one *TC-component* for each $t \in T$, i.e., the resulting overlay is topic-connected.

At each step, an edge $e$ is selected from the potential edge set $E_{pot}$ by **findEdge**() in Line 6 of Alg. 1. Specific algorithms for different TCO problems have their own rules for edge selection, i.e., **findEdge**() is a *virtual* function that needs to be overwritten with an implementation of a concrete criterion, which governs edge selection. We next illustrate these rules for the above listed algorithms. The rules are based on a combination of two criteria: node degree and *edge contribution*, which is defined as reduction in the number of *TC-components* caused by the addition of the edge to the current overlay. The edge contribution for $e$ is denoted as $contrib(e)$.

**Alg. 1** Generalized Overlay Design Algorithm

**Gen-ODA**($V$, $T$, $Int$)

**Input:** $V, T, Int$

**Output:** A topic-connected overlay $TCO(V, T, Int, E)$

1: $E_{new}, E_{pot} \leftarrow \emptyset$
2: **for all** $e{=}(v, w)$ s.t. $(w, v) \notin E_{pot}$ where $v, w \in V$ **do**
3: add $e$ to $E_{pot}$
4: **initDataStructures**()
5: **while** $G(V, E_{new})$ is not topic-connected **do**
6: $e \leftarrow$ **findEdge**()
7: $E_{new} \leftarrow E_{new} \cup \{e\}$
8: $E_{pot} \leftarrow E_{pot} - \{e\}$
9: **updateDataStructures**($e$)
10: return $TCO(V, T, Int, E_{new})$

1. Chockler *et al.* [13] use the **GM-rule** for edge selection with regard to MinAvg-TCO: GM greedily selects an edge with the highest contribution (regardless of the node degree). An optimized implementation of GM has the runtime of $O(|V|^2|T|)$. GM achieves a logarithmic approximation ratio for the average node degree; however, GM only provides an approximation ratio of $\Theta(|V|)$ for the maximum node degree [24].

2. Onus *et al.* [24] use the **MinMax-ODA-rule** for edge selection with regard to MinMax-TCO: MinMax-ODA also selects the edge with the highest contribution, but only among the edges that would minimally increase the maximum node degree. MinMax-ODA always produces a TCO that has a maximum node degree within at most $\log(|V||T|)$ times the optimal maximum node degree.However, MinMax-ODA only attains an approximation ratio of $\Theta(|V|)$ for the average node degree [25].

3. Onus *et al.* [25] propose the **Low-ODA-rule** for solving the Low-TCO problem: Low-ODA uses a parameter $k$ to trade off the balance between average and maximum node degrees. The algorithm makes a weighed selection between the edge $e_1$ chosen by the **GM-rule** and the edge $e_2$ selected by the **MinMax-ODA-rule**: If $contrib(e_1)$ is greater than $k \cdot contrib(e_2)$, $e_1$ is added; otherwise $e_2$ is added. Low-ODA achieves sub-linear approximation ratios on both average and maximum node degrees.

Both MinMax-ODA and Low-ODA find an edge in $O(|V|^2)$ time by scanning all potential edges in a brute force manner, which leads to the time complexity of $O(|V|^4|T|)$ [24,25]. This runtime cost is the main impediment for deploying the algorithms in a relatively static cluster environment where the large degree of control

makes a centralized overlay construction feasible. Furthermore, it limits the scale of validation for MinMax-ODA and Low-ODA which in turn diminishes the potential for using these algorithms as the building blocks (e.g., in the design of divide and conquer algorithms [12]) and the comparison baselines for distributed alternatives.

## 5 Fast Implementation of TCO Algorithms

This section offers an efficient implementation for our proposed Gen-ODA algorithm pattern and its various instantiations. With Alg. 1 as the common pattern, functions **initDataStructures()** and **updateDataStructures()** are shared by different instantiations of Gen-ODA, while **findEdge()** is specialized for different edge selection rules. The fast implementation is based on the new indexing structure that we introduce in this work. A simpler structure was used in [13], which only provided indexing by the edge contribution. In contrast, the structure we propose in this work allows for indexing both by the edge contribution and node degree. In particular, the use of this structure allows us to implement a faster version of MinMax-ODA and Low-ODA running in $O(|V|^2|T|)$ time. By using this faster version, we can accelerate the efficiency of divide-and-conquer algorithms proposed in [12].

We first present the central data structures and elementary functions utilized in our fast implementation of Gen-ODA. Then, we describe the implementation of functions **initDataStructures()** and **updateDataStructures()** shared by different instantiations. Finally, we show how to realize different edge selection rules under the umbrella of this common algorithm pattern. We prove results about the runtime complexity for each of these elements, which allows us to derive the total complexity of $O(|V|^2|T|)$. Table 1 summarizes the overlay construction problems and algorithms, which will be discussed in this section.

Table 1: Algorithms for Solving the TCO Problems

| | |
|---|---|
| **MinAvg-TCO** | **Minimum Average Degree TCO Problem** |
| GM | Greedy Merge algorithm [13], $O(\|V\|^2\|T\|)$ |
| F-MinAvg-ODA | Fast implementation for GM, $O(\|V\|^2\|T\|)$ |
| **MinMax-TCO** | **Minimum Maximum Degree TCO Problem** |
| MinMax-ODA | Minimum Maximum Degree Overlay Design Algorithm [24], $O(\|V\|^4\|T\|)$ |
| F-MinMax-ODA | Fast MinMax-ODA, $O(\|V\|^2\|T\|)$ |
| **Low-TCO** | **Low Avg and Max Degree TCO Problem** |
| Low-ODA | Low Degree Overlay Design Algorithm [25], $O(\|V\|^4\|T\|)$ |
| F-Low-ODA | Fast Low-ODA, $O(\|V\|^2\|T\|)$ |
| $TCO_{\mathsf{ALG}}{}^{*}$ | The TCO produced by ALG |
| $\mathbb{T}_{\mathsf{ALG}}$ | Running time of ALG |

* where ALG stands for any of the discussed algorithms.

### 5.1 An Indexing Data Structure

We introduce an indexing data structure, *EdgeContrib*, as the underlying bedrock for our fast implementation of Gen-ODA. We opt to present *EdgeContrib* in Class 2 using object-oriented design principles, because: (1) it provides a standard interface that can be reused efficiently to develop key functions of Gen-ODA; and (2) the grouping of data and procedures facilitates reasoning about the algorithms and time complexity.

*EdgeContrib* defines an internal class EDGESTRUCT, which encapsulates an edge and meta-information about it, such as its contribution. Besides, EDGESTRUCT contains pointer fields *prev* and *next* to allow inclusion into a doubly-linked list.

*EdgeContrib* contains two additional member attributes: `edgeArray` and `edgeMap`. As illustrated in Fig. 1(a), member `edgeArray` is a 2-dimensional array of size $|T|\times|V|$, which is designed for quick search for the "best" edge at each iteration of Alg. 1. Each entry `edgeArray`$[c][d]$ contains a doubly-linked list of EDGESTRUCT objects corresponding to different edges with contribution $c$ and higher node degree $d$.

Member `edgeMap` is a hashtable such that given an edge $e$, `edgeMap` allows for an efficient lookup of the corresponding EDGESTRUCT$(e)$. In a well-dimensioned hashtable, arbitrary insertions, lookups and deletions have a constant average time cost per operation.

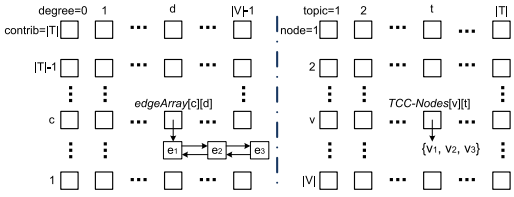

Fig. 1: (a) *EdgeContrib*.`edgeArray` (b) *TCC-Nodes*

**Class 2** *EdgeContrib* Interface and Implementation

// Definition of EDGESTRUCT - data structure for *EdgeContrib* entries

EDGESTRUCT: an encapsulation of an edge and its corresponding information. It is implemented as an element in a doubly-linked list, so that inserting and deleting an edge can be performed in constant time.

- $e(v, w)$: the edge
- $contrib$: the edge contribution of $e(v, w)$, i.e., $contrib(e)$
- $degree$: $\max\{deg(v), deg(w)\}$ where $deg(v)$ is the degree of node $v$ in $G(V, E_{new})$
- $prev$: pointer to its predecessor in the linked list
- $next$: pointer to its successor in the linked list

// Member attributes and auxiliary variables for *EdgeContrib*

▷ `edgeArray`: a 2-dimensional array with $|T|\times|V|$ entries, each representing a set of edges (and their corresponding information) chosen from $V\times V$. An edge $e(v, w)$ is wrapped in an EDGESTRUCT object (see the data structure definition above), denoted as EDGESTRUCT$(e)$, when storing in an entry of `edgeArray`. If EDGESTRUCT$(e) \in$ `edgeArray`$[c][d]$, then: (1) $e \in E_{pot}$; (2) $c =$ EDGESTRUCT$(e).contrib$; (3) $d =$ EDGESTRUCT$(e).degree$.

▷ `edgeMap`: A hashtable that maps an edge $e$ (as a key) to its associated EDGESTRUCT$(e)$ (as a value) in `edgeArray`.

// Functions for *EdgeContrib*

▶ `initEntry`$(c, d)$
1: `edgeArray`$[c][d] \leftarrow \emptyset$

▶ `insertEdge`$(e(v, w), c, d)$
1: construct EDGESTRUCT$(e)$ s.t. $contrib{=}c$ and $degree{=}d$
2: put key-value pair $(e,$ EDGESTRUCT$(e))$ into `edgeMap`
3: insert EDGESTRUCT$(e)$ into `edgeArray`$[c][d]$

▶ `deleteEdge`$(e(v, w), c, d)$
1: get EDGESTRUCT$(e)$ from `edgeMap` using $e$ as the key
2: delete key-value pair $(e,$ EDGESTRUCT$(e))$ from `edgeMap`
3: delete EDGESTRUCT$(e)$ from `edgeArray`$[c][d]$

▶ `getOneEdge`$(c, d)$
1: return the first edge from `edgeArray`$[c][d]$

▶ `getContrib`$(e(v, w))$
1: get EDGESTRUCT$(e)$ from `edgeMap` by key $e$
2: return EDGESTRUCT$(e).contrib$

▶ `getDegree`$(e(v, w))$
1: get EDGESTRUCT$(e)$ from `edgeMap` by key $e$
2: return EDGESTRUCT$(e).degree$

▶ `entrySize`$(c, d)$
1: return $|$`edgeArray`$[c][d]|$

While the implementation of individual functions in *EdgeContrib* is rather straightforward, it is important to observe that each function has a per-invocation runtime cost of $O(1)$. Edge addition or deletion takes constant time thanks to the use of a doubly-linked list. Edge lookup takes $O(1)$ due to using the `edgeMap` hashtable. This property of the constant per-invocation cost is essential for the time efficiency of updating all EDGESTRUCTs in `edgeArray` after adding each edge to the overlay, as we further elaborate upon in Lemma 3.

### 5.2 A Common Template for Implementations

We have showed the outline of Gen-ODA in Alg. 1. A more detailed description with actual data structures for Gen-ODA is presented in the following algorithms: definitions of data structures (Alg. 3), initialization of data structures (Alg. 4) and the update of data structures after each edge addition (Alg. 5). GM [13], MinMax-ODA [24] and Low-ODA [25] all fit into the framework of the Gen-ODA, and the only difference is that they use different criteria to select an edge at each iteration (Line 6 of Alg. 1).

**Alg. 3** Global Variables

- ▶ $EdgeContrib$: an indexing data structure designed for quick search for the best candidate edge using various edge selection rules. See Class 2.
- ▶ $TCC\text{-}Nodes$: a 2-dimensional array of size $|V|\times|T|$ in which each element $TCC\text{-}Nodes[v][t]$ is a subset of $V$ s.t. for each $w \in TCC\text{-}Nodes[v][t]$, (1) $Int(w,t) =$ *true*, and (2) both $w$ and $v$ belong to the same *TC-component* for $t$.
- ▶ $E_{new}$: set of edges in the overlay built so far.
- ▶ $E_{pot}$: set of potential edges that can be added.
- ▶ $nodeDegree$: an array with length $|V|$ s.t. $nodeDegree[v]$ is the degree of node $v$ in $G(V, E_{new})$.
- ▶ $maxContrib$: the highest edge contribution in $E_{pot}$.
- ▶ $maxDegree$: the maximum node degree in $G(V, E_{new})$.
- ▶ $curContrib$: contribution of the currently selected edge.
- ▶ $curDegree$: the higher node degree of the currently selected edge.

Our implementation of Gen-ODA uses several global variables defined in Alg. 3. Among these data structures, $EdgeContrib$ and $TCC\text{-}Nodes$ play the most important roles (see Fig. 1). $EdgeContrib$ is an indexing data structure designed to organize all potential edges (see Class 2). $TCC\text{-}Nodes$ is a 2-dimensional array of size $|V|\times|T|$ which keeps track of the *TC-components* in the current overlay $G(V, E_{new})$: $TCC\text{-}Nodes[v][t]$ holds the set of nodes belonging to the same *TC-component* for $t$ as $v$. To support all these variables for Gen-ODA, a polynomial space is sufficient.

**Alg. 4** Data Structure Initialization

**initDataStructures**()

1: **for all** $v \in V$ **do**
2: $\quad nodeDegree[v] \leftarrow 0$
3: **for all** $v \in V \wedge t \in T$ such that $Int(v,t)$ **do**
4: $\quad TCC\text{-}Nodes[v][t] \leftarrow \{v\}$
5: **for** $c \leftarrow |T|$ **down to** 1 **do**
6: $\quad$ **for** $d \leftarrow 0$ **to** $|V|-1$ **do**
7: $\quad\quad EdgeContrib.\texttt{initEntry}(c,d)$
8: **for all** $e=(v,w) \in E_{pot}$ **do**
9: $\quad c \leftarrow |\{t \in T | Int(v,t) \wedge Int(w,t)\}|$
10: $\quad$ **if** $c > 0$ **then**
11: $\quad\quad EdgeContrib.\texttt{insertEdge}(e,c,0)$
12: $maxContrib$
$\quad \leftarrow \max\{c \mid \exists\, d \text{ s.t. } EdgeContrib.\texttt{entrySize}(c,d)>0\}$
13: $curContrib \leftarrow maxContrib$
14: $curDegree \leftarrow 0, maxDegree \leftarrow 0$

**Lemma 1.** *Alg. 1 takes* $O(|V|^2|T|)$ *space.*

The initialization of these data structures (Alg. 4) takes place at the very beginning of the Gen-ODA algorithm. Gen-ODA starts with the overlay $G(V,\emptyset)$, and Alg. 4 initializes all global variables defined in Alg. 3 accordingly. Lemma 2 shows the time complexity of the initialization.

**Lemma 2.** *The running time of Alg. 4 is* $O(|V|^2|T|)$.

*Proof.* The cost of Gen-ODA's initialization is dominated by the calculation of edge contribution for all potential edges $E_{pot}$ in Lines 8-11 of Alg. 4. If the interest of each node is stored as a list of topics, then the complexity of this computation for $E_{pot}$ will be $O(\sum_{e=(v,w)\in E_{pot}} |\{t \in T | Int(v,t) \wedge Int(w,t)\}|) = O(|V|^2|T|)$. □

After adding $e$ to the overlay and removing it from the potential set (Line 7-8 in Alg. 1), nodes and edges ought to be re-arranged in $EdgeContrib$ and $TCC\text{-}Nodes$ dynamically to reflect the new edge contributions, *TC-components*, and node degrees (Line 9 in Alg. 1). This is performed by Alg. 5.

**Alg. 5** Data Structure Update

**updateDataStructures**($e(v, w)$)

// (1) Update variables for current edge

1: $curContrib \leftarrow EdgeContrib$.getContrib($e$)
2: $curDegree \leftarrow EdgeContrib$.getDegree($e$)
3: $EdgeContrib$.deleteEdge($e$, $curContrib$, $curDegree$)

// (2) Update contributions and *TC-components*

4: **for all** $t \in T$ such that $Int(v, t) \wedge Int(w, t) \wedge TCC\text{-}Nodes[v][t] \neq TCC\text{-}Nodes[w][t]$ **do**
5: **for all** $v' \in TCC\text{-}Nodes[v][t] \wedge w' \in TCC\text{-}Nodes[w][t] \wedge e'(v', w') \neq e(v, w)$ **do**
6: $c \leftarrow EdgeContrib$.getContrib($e'$), $d \leftarrow EdgeContrib$.getDegree($e'$)
7: $EdgeContrib$.deleteEdge($e', c, d$)
8: **if** $c > 1$ **then**
9: $EdgeContrib$.insertEdge($e', c-1, d$)
10: **else**
11: delete $e'$ from $E_{pot}$
12: $new\_tcc\_nodes \leftarrow TCC\text{-}Nodes[v][t] \cup TCC\text{-}Nodes[w][t]$
13: **for all** $u \in new\_tcc\_nodes$ **do**
14: $TCC\text{-}Nodes[u][t] \leftarrow new\_tcc\_nodes$

// (3) Update node degrees

15: $nodeDegree[v] \leftarrow nodeDegree[v] + 1$, $nodeDegree[w] \leftarrow nodeDegree[w] + 1$
16: $maxDegree \leftarrow \max\{maxDegree, nodeDegree[v], nodeDegree[w]\}$
17: **for all** $(v', w') \in E_{pot}$ that is incident on $v$ or $w$ **do**
18: $d_{old} \leftarrow EdgeContrib$.getDegree($(v', w')$)
19: $d_{new} \leftarrow \max\{nodeDegree[v'], nodeDegree[w']\}$
20: **if** $d_{old} < d_{new}$ **then**
21: $c \leftarrow EdgeContrib$.getContrib($(v', w')$)
22: $EdgeContrib$.deleteEdge($(v', w'), c, d_{old}$)
23: $EdgeContrib$.insertEdge($(v', w'), c, d_{new}$)

// (4) Update $maxContrib$

24: **while** $maxContrib > 0$ **do**
25: **for** $degree \leftarrow 0$ **to** $maxDegree$ **do**
26: **if** $EdgeContrib$.entrySize($maxContrib$, $degree$) $> 0$ **then**
27: break from while loop in Line 24
28: $maxContrib \leftarrow maxContrib - 1$

As shown In Alg. 5, the update has four parts: (1) Lines 1-2 update $curContrib$ and $curDegree$ using the currently selected edge; (2) Lines 4-14 update edge contributions for EdgeStructs stored in $EdgeContrib$ and *TC-components* recorded in $TCC\text{-}Nodes$; (3) Lines 17-23 update the array entries in $EdgeContrib$ according to node degrees of $G(V, E_{new})$; (4) Lines 24-28 update the global variable $maxContrib$.

Part (1) and Part (4) deal with basic data types, and are relatively straightforward. Parts (2) and Part (3) are responsible for handling complex data structures.

In Part (2), Lines 6-11 update the contribution of each edge affected by the addition of $e(v, w)$ to the overlay. An edge is affected if its endpoints belong to different *TC-components* prior to the addition but those components are merged as a result of the addition. Once edge $e(v, w)$ is added to the overlay, two *TC-components* are merged into a single one $new\_tcc\_nodes = TCC\text{-}Nodes[v][t] \cup TCC\text{-}Nodes[w][t]$ (Lines 12). Accordingly, for each node $u \in new\_tcc\_nodes$, $TCC\text{-}Nodes[u][t]$ is updated (Lines 13-14).

In Part (3), Alg. 5 handles the node degree update. Lines 15-16 update global variables $nodeDegree$ and $maxDegree$ following the addition of a new edge $e(v, w)$. Lines 17-23 examine all potential edges incident on either $v$ or $w$ and update the corresponding node degrees as the dimension in $EdgeContrib.edgeArray$. For each edge $e'(v', w')$, Line 18 retrieves the *old* degree as the index in $EdgeContrib.edgeArray$, and Line 19 computes the *new* degree in $G(V, E_{new})$; Lines 20-23 update the indexing structure if $d_{old} < d_{new}$.

Lemma 3 shows the cumulative running time of updates performed by Alg. 5 for all edges added to the TCO.

**Lemma 3.** *The cumulative running time of all invocations of Alg. 5 during the entire execution of Alg. 1 is* $O(|V|^2|T|)$.

*Proof.* The runtime cost of updates in Alg. 5, invoked after adding an edge, is dominated by Part (2) and Part (3).

When updating a contribution or a degree for an edge, its corresponding EDGESTRUCT entry can be found in average $O(1)$ time in $EdgeContrib.edgeArray$ with the assistance of $EdgeContrib.edgeMap$ (see Class 2 in Sec. 5.1). The update of each individual EDGESTRUCT in $EdgeContrib$ can be performed in $O(1)$ by constant time operations of $EdgeContrib.\texttt{deleteEdge}()$ and $EdgeContrib.\texttt{insertEdge}()$.

In Part (2), in order to calculate the total count of individual edge updates at all iterations, it is sufficient to notice that every update decrements the contribution of the edge by one (Lines 6-11). Alg. 1 starts when the total contribution of all edges is $O(\sum_{e=(v,w)\in E_{pot}} |\{t \in T | Int(v,t) \wedge Int(w,t)\}|) = O(|V|^2|T|)$, and terminates when the contribution of all the edges is reduced to zero. Therefore, the cumulative cost of updating all edge contributions is $O(|V|^2|T|)$.

In Part (3), the update of node degrees when adding an edge is bounded by $O(|V|)$, and the size of the output edge set is at most $O(\min\{|V||T|, |V|^2\})$, so the overall cost of updating node degrees is $O(\min\{|V|^2|T|, |V|^3\})$.

In summary, the cumulative runtime cost of updates for all edges added to the overlay is $O(|V|^2|T|)$. □

Having presented an efficient implementation of **initDataStructures**() and **updateDataStructures**() for Alg. 1, we now focus on the concrete realizations of **findEdge**() for different TCO construction criteria in Sec. 5.3, 5.4, and 5.5. At each iteration of Gen-ODA, **findEdge**() (Line 6 in Alg. 1) finds an edge $e$, whose addition would merge at least two different *TC-components* (for at least one topic), thus reducing the total number of *TC-components* by at least one. While naive search for the next "best" edge takes $O(|V|^2)$ time, the implementation presented here improves the time complexity by employing the auxiliary indexing data structure $EdgeContrib$. This data structure facilitates finding the 'best' edge at each iteration taking both edge contribution and node degree into account because the algorithm can traverse $EdgeContrib.edgeArray[c][d]$ in the order of decreasing contribution $c$ and increasing degree $d$ and pick an edge from the first non-empty entry.

### 5.3 Finding Edge for MinAvg-TCO

Gen-ODA together with Alg. 6, referred to as F-MinAvg-ODA (Fast MinAvg Overlay Design Algorithm), builds the same overlay as GM [13]. Alg. 6 implements the **GM-rule**: it always chooses the edge with the highest contribution toward topic-connectivity regardless of node degrees.

**Alg. 6** Find a MinAvg Edge

**findMinAvgEdge()**

**Output:** an edge $e$ to be added to $E_{new}$

```
1: for degree ← curDegree to maxDegree do
2:     if EdgeContrib.entrySize(maxContrib,
       degree) > 0 then
3:         e←EdgeContrib.getOneEdge(maxContrib,
           degree)
4:         return e
```

Lemma 4 shows that F-MinAvg-ODA achieves the same time efficiency as GM. The formal proof for Lemma 4 is omitted here, since it basically is a simplification of the time efficiency proof for F-MinMax-ODA, which we present in Sec. 5.4.

**Lemma 4.** *The cumulative running time of all invocations of Alg. 6 during the entire execution of Alg. 1 is* $O(|V|^2|T|)$.

### 5.4 Finding Edge for MinMax-TCO

MinMax-ODA in [24] yields the time complexity of $O(|V|^4|T|)$. Gen-ODA with the **MinMax-ODA-rule** implemented in Alg. 7 provides an efficient realization of MinMax-ODA, with an improved running time of $O(|V|^2|T|)$. We refer to this combined algorithm as F-MinMax-ODA (Fast MinMax-ODA).

In order to explain Alg. 7, we first observe that MinMax-ODA (and consequently F-MinMax-ODA) adds new edges in phases. At the start of each phase, MinMax-ODA selects a new edge that increases the maximum degree of the overlay by one. Then, the algorithm proceeds with adding edges without raising the maximum degree until the addition of any extra edge would cause a new increase, at which point the phase ends. The number of such phases is limited by the highest possible overlay degree, i.e., $O(|V|)$.

When invoked by Alg. 1 at each iteration, Alg. 7 scans the entries corresponding to non-maximum degree ($< maxDegree$) in `edgeArray` of $EdgeContrib$ in the order of increasing degree and decreasing contribution. If a non-empty entry is found, an arbitrary edge from the entry edge list is selected. Otherwise, an edge from the entry with the maximum contribution and maximum degree is selected, which leads to the increase in the overlay degree and signifies a start of a new phase.

**Alg. 7** Find a MinMax Edge

```
findMinMaxEdge()
Output: an edge e to be added to E_new
1:  e ← NIL, contrib ← curContrib
2:  while e = NIL ∧ contrib > 0 do
3:      initDegree ← 0
4:      if contrib = curContrib then
5:          initDegree ← curDegree
6:      for degree←initDegree to maxDegree−1 do
7:          if EdgeContrib.entrySize(contrib,
            degree) > 0 then
8:              e←EdgeContrib.getOneEdge(contrib,
                degree)
9:              break from for loop in Line 6
10:     contrib ← contrib − 1
11: if e = NIL then
12:     e←EdgeContrib.getOneEdge(maxContrib,
        maxDegree)
13: return e
```

The crucial element for the efficiency of the implementation is that rather than scanning the entire `edgeArray` of $EdgeContrib$ at each invocation, Alg. 7 continues the scan from the last selected entry. First, it does not affect the correctness of the scan: while after an edge addition, Alg. 5 reshuffles potential edges across `edgeArray`, it only moves the edges in the order of decreasing *contrib* (Lines 7-9) or increasing *degree* (Lines 22-23). Since Alg. 7 scans the entries in precisely the same order, it cannot miss a potential edge.

Secondly, continuing the scan from the last selected entry upon each Alg. 7 invocation within a single phase implies that the number of entries scanned at each phase is limited by the sum of two factors: the total number of entries in `edgeArray` of $EdgeContrib$ (which is equal to $|V| \cdot |T|$) plus the number of entries scanned multiple times, i.e., the number of Alg. 7 invocations, which is equal to the number of

edges selected at this phase (which is limited by $\frac{|V|}{2}$ [24]). Therefore, the number of entries scanned during the entire execution of Alg. 1 (i.e., at all $O(|V|)$ phases) is $O(|V| \cdot (|V||T| + \frac{|V|}{2})) = O(|V|^2|T|)$. This underlines the proof of Lemma 5; we provide the complete proof below.

**Lemma 5.** *The cumulative running time for all invocations of Alg. 7 during the entire execution of Alg. 1 is* $O(|V|^2|T|)$.

*Proof.* Suppose that $TCO_{\mathsf{FMM}}(V, T, Int, E_{\mathsf{FMM}})$ is the overlay network produced by F-MinMax-ODA, and $D_{\mathsf{FMM}}$ is the maximum node degree in $TCO_{\mathsf{FMM}}$. As shown in Sec. 5.4, F-MinMax-ODA works in phases. We denote $M_i$ as the edge set added to $E_{\mathsf{FMM}}$ by F-MinMax-ODA at phase $i$. At the start of the $i$-th phase, F-MinMax-ODA selects an edge that increases $maxDegree$ from $i-1$ to $i$, then the algorithm proceeds by adding edges that do not raise $maxDegree$. Phase $i$ ends and phase $i+1$ starts when the addition of any edge would increase $maxDegree$ from $i$ to $i+1$.

At each invocation by Line 6 of Alg. 1, Alg. 7 finds an edge for F-MinMax-ODA. The algorithm first searches for a non-empty entry in $EdgeContrib.\texttt{edgeArray}$, and then picks an edge from that entry. We consider all invocations of Alg. 7 for each phase. Using amortized analysis we show that the cumulative runtime cost is bounded by $O(|V||T|)$.

More specifically, the cumulative runtime cost of all invocations of Alg. 7 at the $i$-th phase is dominated by two components:

(1) The cumulative cost of searching for a non-empty entry in $\texttt{edgeArray}$ of $EdgeContrib$: this is determined by the number of entry probing operations in Line 7 of Alg. 7. In fact, although a single invocation of Alg. 7 can probe multiple entries, the number of probing operations for all edges in $M_i$ is bounded by the sum of two factors: 1) the total number of times the probed entry turns out to be empty so that Line 7 of Alg. 7 return *false*, and 2) the total number of times Line 7 of Alg. 7 return *true*. Factor 1) is bounded by the total number of entries in $EdgeContrib.\texttt{edgeArray}$, i.e., $|V| \cdot |T|$. This is because: at each iteration within the $i$-th phase of adding $M_i$, we always start probing from the last selected entry. The pointer for array entry probing only moves in one direction: in the order of increasing $degree$ or decreasing $contrib$. The probing never moves backwards, so entries that have been checked as empty would not be visited again in this phase. Factor 2) is bounded by the number of edges selected in this phase, which is not higher than $\frac{|V|}{2}$ [24]. In summary, the cumulative running time of searching for a non-empty entry in the $i$-th phase is $O(|V||T|)$.

(2) The cumulative cost of picking an edge from the identified non-empty entry: this is determined by the number edge selection operations in Line 8 of Alg. 7. It takes $O(1)$ to obtain an edge by calling $EdgeContrib.\texttt{getOneEdge}()$. Totally there are at most $\frac{|V|}{2}$ edges to be added in each phase [24]. Thus, the cumulative running time of edge picking in the $i$-th phase is $O(|V|)$.

To sum up, there are at most $O(|V|)$ phases, so the cumulative cost of all invocations of Alg. 7 during one entire execution of Alg. 1 is $O(|V|) \cdot (O(|V||T|) + O(|V|)) = O(|V|^2|T|)$. □

### 5.5 Finding Edge for Low-TCO

A naive implementation of Low-ODA yields the time complexity of $O(|V|^4|T|)$ (see Lemma 3 in [25]). The Gen-ODA implementing the **Low-ODA-rule** is described in Alg. 8, which we refer to as F-Low-ODA (Fast Low-ODA), produces the same overlay with the improved running time of $O(|V|^2|T|)$. Combined, Lemma 4 and Lemma 5 allow us to establish Lemma 6.

**Alg. 8** Find a Low Edge

**findLowEdge**($k$)

**Input:** $k$: parameter to balance edge selection rules
**Output:** an edge $e$ to be added to $E_{new}$

1: $e_1 \leftarrow$ **findMinAvgEdge**(), $contrib_1 \leftarrow EdgeContrib$.getContrib($e_1$)
2: $e_2 \leftarrow$ **findMinMaxEdge**(), $contrib_2 \leftarrow EdgeContrib$.getContrib($e_2$)
3: **if** $contrib_1 \geq contrib_2 \times k$ **then**
4: return $e_1$
5: **else**
6: return $e_2$

**Lemma 6.** *The cumulative running time for all invocations of Alg. 8 during the entire execution of Alg. 1 is* $O(|V|^2|T|)$.

### 5.6 Running Time for Gen-ODA

To summarize all complexity analyses based on Lemmas 2, 3, 4, 5 and 6, the following lemma establishes the time efficiency of our implementation for F-MinAvg-ODA, F-MinMax-ODA and F-Low-ODA.

**Lemma 7.** *The running time of Alg. 1 with function* ***findEdge****() instantiated as either Alg. 6, Alg. 7 or Alg. 8 is* $O(|V|^2|T|)$.

## 6 Evaluation

We implement all algorithms in Table 1 in Java and evaluate the running time of different algorithms, i.e., F-MinMax-ODA (vs. MinMax-ODA) and F-Low-ODA (vs. Low-ODA). We denote by $T_v$ the topic set which node $v$ subscribes to, and by $|T_v|$ the *subscription size* of node $v$. In these experiments, we use the following value ranges as input: $|V| \in [100, 1\,000]$, $|T| \in [100, 1\,000]$, and $|T_v| \in [10, 100]$, where the subscription size is fixed for each node in the input. Each topic $t_i \in T$ is associated with probability $q_i$, $\sum_i q_i = 1$, so that each node subscribes to $t_i$ with a probability $q_i$. The value of $q_i$ is distributed according to either a uniform, a Zipf (with $\alpha = 2.0$), or an exponential distribution. According to [14], these distributions are representative of actual workloads used in industrial pub/sub systems today. Liu *et al.* [23] show that the Zipf distribution faithfully describes the feed popularity distribution in RSS feeds (a pub/sub-like application scenario). The exponential distribution is used by stock-market monitoring engines for the study of stock popularity in the New York Stock Exchange [29].

### 6.1 F-MinMax-ODA for MinMax-TCO

We now consider F-MinMax-ODA's performance compared to MinMax-ODA with respect to different input parameters. Both F-MinMax-ODA and MinMax-ODA algorithms use the **MinMax-ODA-rule** for edge selection but are based on different implementations. Since the TCOs they compute are the same, we only show their running time ratios here.

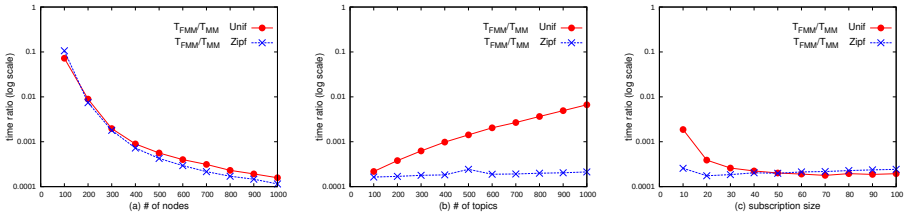

Fig. 2: F-MinMax-ODA vs. MinMax-ODA

Fig. 2(a) depicts the comparison between F-MinMax-ODA and MinMax-ODA as the number of nodes increases when $|T| = 100$. As the figure shows, F-MinMax-ODA runs considerably faster. Under uniform distribution, $\mathbb{T}_{\mathsf{FMM}}$ is on average $0.858\%$ of $\mathbb{T}_{\mathsf{MM}}$; under Zipf distribution, $\mathbb{T}_{\mathsf{FMM}}$ is on average $1.17\%$ of $\mathbb{T}_{\mathsf{MM}}$. Additionally, the F-MinMax-ODA algorithm gains more speedup with the increase in the number of nodes compared to MinMax-ODA: when $|V|$=1000, $\mathbb{T}_{\mathsf{FMM}} = 0.0158\% \cdot \mathbb{T}_{\mathsf{MM}}$ for the uniform distribution and $\mathbb{T}_{\mathsf{FMM}} = 0.0115\% \cdot \mathbb{T}_{\mathsf{MM}}$ for the Zipf distribution. The gap in the running time between our algorithms and existing ones is so significant that instead of showing the absolute values on the same scale we opt to present the ratio. For example, under the Zipf distribution, with 1000 nodes and 100 topics, F-MinMax-ODA completes in $3.823$ seconds, while MinMax-ODA takes over $555$ minutes. This shows that F-MinMax-ODA provides an adequate solution for the above target settings while MinMax-ODA does not.

Fig. 2(b) depicts how F-MinMax-ODA and MinMax-ODA perform when the number of topics varies. The running time ratio of F-MinMax-ODA to MinMax-ODA increases as the number of topics increases from 100 to 1000. In order to explain this effect, we observe that the running time of scanning the indexing structure in F-MinMax-ODA is proportional to the maximum edge contribution while the running time of MinMax-ODA is independent of edge contributions. Increasing the number of topics leads to reduced correlation, i.e., the probability of having two nodes interested in the same topic drops as the number of topics increases, and with reduced correlation the edge contribution tends to be lower. This reduction in correlation is more pronounced for the uniform distribution of interests compared to skewed ones, such as Zipf. Yet, the increase in the running time ratio is not very significant: on average, F-MinMax-ODA is less than $0.236\%$ of MinMax-ODA under the uniform distribution, and less than $0.019\%$ under the Zipf distribution.

Fig. 2(c) depicts the impacts of the subscription size on F-MinMax-ODA and MinMax-ODA. We set $|T| = 200$, and $|T_v|$ varies from 10 to 100. As shown in the figure, the ratio of $\mathbb{T}_{\mathsf{FMM}}$ to $\mathbb{T}_{\mathsf{MM}}$ decreases with the increase of $|T_v|$, and the ratio becomes relatively stable around $0.02\%$ when $|T_v| > 50$.

### 6.2 F-Low-ODA for Low-TCO

We now explore the impact of different input variables on the performance of the F-Low-ODA and Low-ODA algorithms. Both apply the **Low-ODA-rule** for edge selection, so for the evaluation, we only consider their implementation efficiency.

Fig. 3(a) depicts the comparison between these two algorithms as the number of nodes increases where $|T|$=100. As the figure shows, F-Low-ODA runs significantly faster. Under the uniform distribution, $\mathbb{T}_{\mathsf{FLOW}}$ is on average $1.2\%$ of $\mathbb{T}_{\mathsf{LOW}}$. Under the

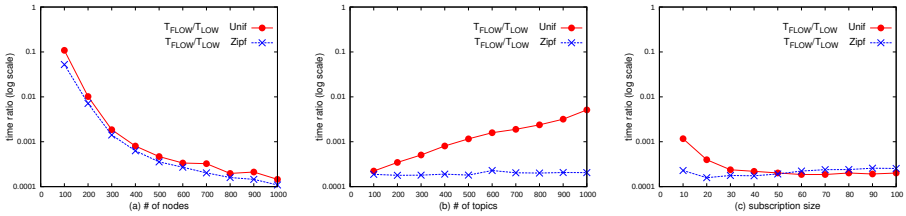

Fig. 3: F-Low-ODA vs. Low-ODA

Zipf distribution, $\mathbb{T}_{\text{FLOW}}$ is on average $0.6\%$ of $\mathbb{T}_{\text{LOW}}$. Additionally, F-Low-ODA gains more speedup with the increase in the number of nodes compared to Low-ODA: when $|V| = 1000$, $\mathbb{T}_{\text{FLOW}} = 0.15\% \cdot \mathbb{T}_{\text{LOW}}$ for the uniform distribution and $\mathbb{T}_{\text{FMM}} = 0.11\% \cdot \mathbb{T}_{\text{MM}}$ for the Zipf distribution.

Fig. 3(b) depicts the performance of F-Low-ODA and Low-ODA when we vary the number of topics. The ratio of $\mathbb{T}_{\text{FLOW}}$ to $\mathbb{T}_{\text{LOW}}$ increases as the number of topics increases from 100 to 1000, yet this effect is insignificant: on average, F-Low-ODA takes less than $0.172\%$ of Low-ODA's running time under the uniform distribution and less than $0.020\%$ under the Zipf distribution. Further, F-Low-ODA has more speedup on the time efficiency for skewed distributions as the number of topics increases. The reason is that increasing the number of topics leads to less correlation, and under skewed distribution, the correlation among nodes drops relatively slower compared to that under the uniform distribution.

Fig. 3(c) depicts the effects of the subscription size on F-Low-ODA and Low-ODA. We set $|T|$=200 and $|T_v| \in [10, 100]$. As shown in the figure, the running time ratio $\frac{\mathbb{T}_{\text{FLOW}}}{\mathbb{T}_{\text{LOW}}}$ decreases with the increase of $|T_v|$. The ratio becomes stable around $0.02\%$ as $|T_v| > 50$.

## 7 Conclusions

In this paper, we develop the Gen-ODA framework that covers existing greedy algorithms with different edge selection rules for different optimization criteria. By using the indexing data structures that we have devised, a number of known algorithms gain a significant running time speedup, i.e., the time complexity of MinMax-ODA and Low-ODA is improved from $O(|V|^4|T|)$ to $O(|V|^2|T|)$.

We have evaluated the algorithms through a comprehensive experimental analysis, which demonstrates their performance and scalability under various practical pub/sub workloads. Our proposed Gen-ODA is well suited to different TCO construction problems: its efficient implementation accelerates the time efficiency by a factor of more than 1 000, and it gains more impact in the running time when the workloads scale up.

## References

1. GDSN, http://bit.ly/cjnevk
2. Google Cluster Data, http://code.google.com/p/googleclusterdata/
3. TIBCO Rendezvous, http://www.tibco.com
4. Araujo, F., Rodrigues, L., Carvalho, N.: Scalable QoS-based event routing in publish-subscribe systems. In: NCA (2005)
5. Baehni, E., Eugster, P., Guerraoui, R.: Data-aware multicast. In: DSN (2004)

6. Baldoni, R., Beraldi, R., Quema, V., Querzoni, L., Tucci-Piergiovanni, S.: TERA: topic-based event routing for peer-to-peer architectures. In: DEBS (2007)
7. Baldoni, R., Beraldi, R., Querzoni, L., Virgillito, A.: Efficient publish/subscribe through a self-organizing broker overlay and its application to SIENA. Comput. J. 50(4) (2007)
8. Castro, M., Druschel, P., Kermarrec, A.M., Rowstron, A.: SCRIBE: A large-scale and decentralized application-level multicast infrastructure. JSAC (2002)
9. Chand, R., Felber, P.: Semantic peer-to-peer overlays for publish/subscribe networks. In: EUROPAR (2005)
10. Chen, C., Jacobsen, H.-A., Vitenberg, R.: Divide and conquer algorithms for publish/subscribe overlay design. In: ICDCS (2010)
11. Chen, C., Vitenberg, R., Jacobsen H.-A.: A generalized algorithm for publish/subscribe overlay design and its fast implementation. Tech. rep., U. of Toronto & U. of Oslo, http://msrg.org/papers/TRCVJ-GenODA
12. Chen, C., Vitenberg R., Jacobsen H.-A.: Scaling construction of low fan-out overlays for topic-based publish/subscribe systems. In: ICDCS (2010)
13. Chockler, G., Melamed, R., Tock, Y., Vitenberg, R.: Constructing scalable overlays for pub-sub with many topics: Problems, algorithms, and evaluation. In: PODC (2007)
14. Chockler, G., Melamed, R., Tock, Y., Vitenberg, R.: Spidercast: A scalable interest-aware overlay for topic-based pub/sub communication. In: DEBS (2007)
15. Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P. and Jacobsen, H.-A., Puz, N., Weaver, D., Yerneni, R.: PNUTS: Yahoo!'s hosted data serving platform. Proc. VLDB Endow. (2008)
16. De Santis, E., Grandoni, F., Panconesi, A.: Fast low degree connectivity of ad-hoc networks via percolation. In: Percolation. In: Arge, L., Hoffmann, M., Welzl, E. (eds.) ESA 2007. LNCS, vol. 4698, pp. 206–217. Springer, Heidelberg (2007)
17. Girdzijauskas, S., Chockler, G., Vigfusson, Y., Tock, Y., Melamed, R.: Magnet: practical subscription clustering for internet-scale publish/subscribe. In: DEBS'10
18. Jaeger, M.A., Parzyjegla, H., Mühl, G., Herrmann, K.: Self-organizing broker topologies for publish/subscribe systems. In: SAC (2007)
19. Lau, L.C., Naor, J.S., Salavatipour, M.R., Singh, M.: Survivable network design with degree or order constraints. In: Proc. ACM STOC (2007)
20. Li, G., Muthusamy, V., Jacobsen, H.-A.: Adaptive content-based routing in general overlay topologies. In: Issarny, V., Schantz, R. (eds.) Middleware 2008. LNCS, vol. 5346, pp. 1–21. Springer, Heidelberg (2008)
21. Li, G., Muthusamy V., Jacobsen, H.-A.: A distributed service oriented architecture for business process execution. ACM TWEB (2010)
22. Liben-Nowell, D., Balakrishnan, H., Karger, D.: Analysis of the evolution of peer-to-peer systems. In: PODC (2002)
23. Liu, H., Ramasubramanian, V., Sirer, E.G.: Client behavior and feed characteristics of RSS, a publish-subscribe system for web micronews. In: IMC (2005)
24. Onus, M., Richa, A.W.: Minimum maximum degree publish-subscribe overlay network design. In: INFOCOM (2009)
25. Onus, M., Richa, A.W.: Parameterized maximum and average degree approximation in topic-based publish-subscribe overlay network design. In: ICDCS (2010)
26. Petrovic, M and Liu, H., Jacobsen, H.-A.: G-ToPSS: fast filtering of graph-based metadata. In: WWW (2005)
27. Reumann, J.: Pub/Sub at Google, Lecture & Personal Communications at EuroSys & CANOE Summer School, Oslo, Norway, Aug 2009
28. Tam, D., Azimi, R., Jacobsen H.-A.: Building content-based publish/subscribe systems with distributed hash tables. In: DBISP2P (2003)

29. Tock, Y., Naaman, N., Harpaz, A., Gershinsky, G.: Hierarchical clustering of message flows in a multicast data dissemination system. In: IASTED PDCS (2005)