Compact Implementation of Distributed Inference Algorithms for Network

Ashima Atul

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-39
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-39.html

March 12, 2009

Compact Implementation of Distributed Inference Algorithms for Network Monitoring

by

Ashima Atul

B.E. (Army Institute of Technology, Pune University, India) 2004

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Joseph M. Hellerstein, Chair
Professor Vern Paxson
Professor John Chuang

Fall 2008

The thesis of Ashima Atul is approved.

Professor Joseph M. Hellerstein, Chair Date

Professor Vern Paxson Date

Professor John Chuang Date

University of California, Berkeley

Fall 2008

Compact Implementation of Distributed Inference Algorithms for Network Monitoring

by

Ashima Atul

Abstract

Compact Implementation of Distributed Inference Algorithms for Network Monitoring

by

Ashima Atul

Master of Science in Computer Science

University of California, Berkeley

Professor Joseph M. Hellerstein, Chair

In this thesis we present the compact implementation of distributed inference algorithms using a declarative programming framework. This framework provides a declarative query language for specifying and implementing distributed protocols. We show the practicality of distributed inference algorithms by applying them to network monitoring applications.

There has been a growing trend towards automated generation of massive amounts of data at multiple distributed locations. These systems can take advantage of learning and inference algorithms to assemble local observations and reach global conclusions. This requires performing probabilistic inference in a distributed fashion. Distributed inference algorithms eliminate single points of failure, distribute the computation across several nodes and avoid the need to share sensitive data.

The design and implementation of distributed algorithms is very challenging. Our work involves using a combination of overlays and declarative programming to simplify the design of distributed inference algorithms. Our main contribution involves using a declarative language, Overlog, to implement a set of existing probabilistic inference

algorithms and evaluating their performance. We also present the conciseness of our declarative implementation. For example, we could implement *Junction Tree Running Intersection Property* in just *7 rules* in the Overlog.

We then use the distributed inference architecture for collaborative spam detection. This work is a first step towards applying distributed inference techniques for network monitoring. During the design of the application we learned that multiple factors like algorithm selection, data partitioning and aggregation play an important role when solving network monitoring problems using distributed inference. Future work in this direction involves solving the issues faced to make the spam detection application scalable and practical to provide real-time detection.

Professor Joseph M. Hellerstein, Chair Date

Acknowledgements

I would like to thank my advisor Joe Hellerstein. I am lucky to have got the chance to work with Joe. I started working with him as a volunteer for a project in the P2 Declarative Networking group. After the project he accepted me as one of his students at Berkeley and gave me the opportunity to continue working with the group. Joe contributed countless hours of his precious time to advise me on research, writing, collaboration and job search. His personal attention has been invaluable. Apart from technical guidance he provided practical advice that helped me in getting through graduate school life with little friction. It has been a pleasure working with him.

My work related to distributed inference would not have been possible without the guidance of Prof. Guestrin. Even though I was not Prof. Guestrin's official student, he helped me in understanding many inference algorithms and how to distribute them.

Two students who made significant contribution to the initial work of implementing distributed versions of the existing inference algorithms used in this thesis are Stanislav Funiak and Kuang Chen. I am very glad to have got the chance to work with Stanislav, a student of Prof. Guestrin. He made significant contributions to the codebases that made my research possible. I am also really happy to have had Kuang Chen onboard during the initial phase of the distributed inference project. He helped us in building the basic infrastructure for inference and played an important role in implementing the distributed junction tree algorithm. His contribution is invaluable to this thesis.

I would also like to thank Prof. Feamster and his student Anirudh Ramachandran for helping me understand the schemes they used in their SpamTracker project and also analyzing the spam dataset. Prof. Feamster played a major role in suggesting different network monitoring applications to which inference techniques could be applied. I am also thankful to Udam Saini for helping in the implementation of the collaborative spam detection application.

I would also like to thank P2 team members Tyson condie and Petros Maniatis. Tyson

Condie took out significant time from his schedule to add functionalities to the P2 framework as per our requirements and also helped us in debugging several issues.

It has been wonderful working with the entire database research group. Alexandra Meliou, Russell Sears, David Chu, Daisy Wang, Eirinaios Michelakis, Peter Alvaro, Neil Conway, Beth Trushkowsky and Christine Robson have provided many helpful comments, friendship, and plenty of fun.

I also like to thank Prof. Paxson and Prof. Chuang for participating on my thesis committee. Prof. Paxson gave very valuable insights while I was working on the spam detection project. Moreover, my interest in the field of network security is mainly because of the network security course I took under him.

My family in India has been very supportive of me all these years. Even though they live halfway around the world, my parents, in-laws, brother and sister-in-law have been constant sources of encouragement.

I am deeply indebted to my husband Siddharth for his support and understanding. He encouraged me to pursue higher studies after we got married. He makes sure to point all my mistakes and make me learn from them. I am lucky to have him as my pillar of strength and dedicate this thesis to him.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In recent years, there has been a growing trend towards automated generation of massive amounts of data at multiple distributed locations. The distributed nature of automatically-generated information is present in both the physical world (monitoring temperature through sensors) and in computer networks (network firewall logs, traffic monitoring and spam feature extraction). However, the information gathered by nodes at different locations is typically too restricted to be of direct interest; it must be computationally assembled. For example, domain mail servers in a network may assemble their local observations of the extracted features from received emails in order to identify spammers. Such systems can take advantage of learning and inference algorithms to combine local measurements and provide a globally consistent view. This raises the need for performing probabilistic inference in a distributed fashion. Distributed inference algorithms eliminate single points of failure, distribute the computation across several nodes and avoid the need to share sensitive data [22].

Distributed inference is a challenging task and requires addressing a lot of design issues. First, it is difficult to design a distributed inference algorithm since each node needs to coordinate and distribute the computation across the network, and can only access a

portion of the data at any point of time. Second, the algorithm has to be network aware. It should monitor changes in link quality, node failure and addition, as well as network partitioning. Third, programming distributed systems is itself not an easy task. For example, implementing the system in low-level languages is very error-prone and difficult to debug. Addressing these challenges is key to designing robust and efficient distributed inference algorithms.

1.2 Contribution

In this thesis we evaluate **declarative networking** [26] techniques for implementing distributed inference algorithms. A declarative language provides ease of programming and allows the user to specify at a high level *what* to do, rather than *how* to do it. We believe that declarative programming will help accelerate the development of distributed inference algorithms.

Distributed inference requires nodes in multiple locations to communicate with each other by sending messages. In order to send messages each node needs to know about other nodes and their network address in the underlying network. Instead of using existing network and its addresses we can use an overlay network to form a virtual network of nodes and have logical links between the nodes. Overlay networks route and address messages via an application-specific naming scheme rather than using Internet addressing (for example, distributed hash tables for content based routing [45, 35, 12]) and also provide specific network communication patterns like multicast [20]. Declarative programming languages such as **Overlog** have been used in the past to express overlays [25]. Overlog provides concise implementation of overlays compared to their low-level implementation and also aids in ease of development and deployment.

To realize our vision of distributed inference using the **P2 declarative programming framework**, and its language **Overlog**, we make the following contributions in this thesis:

- We demonstrate the use of declarative programming for implementing distributed

versions of existing inference algorithms, such as **Junction Tree Inference** [31] and **Loopy Belief Propagation** [29, 46]. We show that Overlog programs are a natural and compact way of expressing a variety of well-known inference algorithms, typically in a handful of lines of program code. Overlog also allows ease of customization, where higher-level concepts such as message scheduling can be achieved via simple modifications to the Overlog programs. For example, we customize the naive loopy belief propagation algorithm by adding a randomized message scheduling scheme to achieve faster convergence.

- We evaluate our declarative implementation of junction tree inference on the sensor calibration dataset used by Paskin et al. [31]. Our working implementation of junction tree inference is specified in 37 Overlog rules versus thousands of lines of Lisp code of the distributed implementation by Paskin et al [31].

- To explore applications of distributed inference in the context of network monitoring we take a centralized, offline scheme from the work of Feamster et al. [34] and evaluate whether a distributed implementation of this scheme can be easily built using declarative networking framework. We are able to demonstrate that our distributed implementation can achieve similar results as the centralized scheme, although our implementation is less scalable due to the limitations in the P2 declarative framework and issues faced with the selected inference algorithm. During the course of this exercise the lessons learned include exploiting data partitioning to reduce aggregation, sharing summary statistics instead of data to reduce communication bandwidth and selecting inference algorithms that do not have a high cost of communication to achieve global consistency.

The evaluation of our distributed inference implementations is done on Emulab [47], a large testbed that simulates realistic network conditions such as communication delays.

1.3 Organization

This thesis is organized as follows. In Chapter 2, we provide an introduction to distributed inference and the architecture for designing distributed inference algorithms, and give an overview of Overlog, the declarative language of P2 framework. Chapter 3 presents declarative distributed implementation of well-known inference algorithms. Chapter 4 demonstrates how distributed inference can be used for network monitoring. We take the collaborative spam detection application and use a distributed inference technique to detect spammers. Chapter 5 discusses future directions in terms of using distributed inference for building a scalable spam detection application.

Chapter 2

Distributed Inference

Having discussed the need for distributed inference algorithms, this chapter focuses on the architecture needed to design inference algorithms for performing distributed computation. The layered graph architecture presented in this chapter separates the **communication graph** from the **reasoning graph**. A communication graph represents the communication links between nodes while a reasoning graph represents correlations and independencies in data residing at different nodes and facilitates designing efficient distributed inference algorithms. Separating the communication and reasoning graph aids in optimizing the communication and the reasoning components, to effectively make tradeoffs between resource usage and information quality.

In Section 2.1 we give an overview of probabilistic graphical models. We then present the layered graph architecture for modeling distributed inference in Section 2.2. In section 2.3, we explain the reasoning graph layer, present its usefulness by applying it to a network monitoring problem and discuss the challenges in implementing distributed inference over the reasoning graph. We give an overview of the declarative language, Overlog, used for implementing the algorithms in Section 2.4.

2.1 Probabilistic Graphical Models Overview

In this section, we provide a short overview of probabilistic graphical models, a machine learning technique, that uses graphs to represent correlations and independencies in data and facilitates in designing efficient inference algorithms. We call this graph the **reasoning graph** in our architecture. We explain the reasoning graph in detail in Section 2.2.

A graphical model is a family of probability distributions defined in terms of a graph. The nodes in the graph represent random variables that can be discrete, continuous or a binary event. The joint probability distribution is obtained by taking a product over functions defined on the connected subset of nodes. The graph can be directed or undirected; we explain the undirected case of graphical models here.

Given an undirected graph $G(V, E)$, we have $X_v : v \in V$ as a collection of random variables indexed by nodes in the graph. Let C be the maximal cliques in the graph. Each clique x_c of X_C is associated with a non-negative potential function $\psi_c(x_c)$. An undirected graphical model represents the joint distribution of x_v that can be factored into the product of functions over the variables in each clique:

$$p(x_v) = \frac{1}{Z_C} \prod_{c \in C} \psi_c(x_c). \tag{2.1}$$

where Z is the normalization factor, obtained by integrating and summing the product with respect to x_v (Equation 2.2).

$$Z_C = \sum_{x_v} \left\{ \prod_{c \in C} \psi_c(x_c) \right\}. \tag{2.2}$$

Figure 2.1 gives an example of undirected probabilistic model [1].

2.2 Graphs Upon Graphs

Our approach to model distributed inference algorithms uses a layered graph architecture. The layered architecture has two types of graphs (shown in Figure 2.2):

[1] This example has been taken from *An Introduction to Probabilistic Graphical Models* by M. I. Jordan

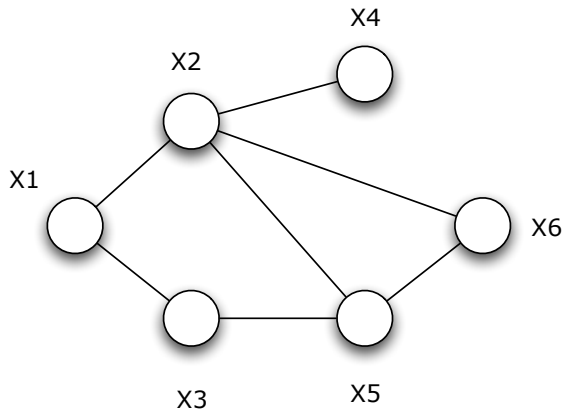


Figure 2.1. An example of an undirected graphical model. The probability distribution associated with this graph can be factorized as $p(x_v) = \frac{1}{Z_C}\psi_(x_1, x_2)\psi_(x_1, x_3)\psi_(x_2, x_4)\psi_(x_3, x_5)\psi_(x_2, x_5, x_6)$.

1. A **communication overlay graph** that handles the communication between nodes distributed at multiple locations.

2. A **reasoning graph** that probabilistically characterizes how locally observed information is combined to reach global conclusions.

In this architecture, the information resides in nodes at multiple locations. These nodes can be sensors monitoring temperature in a sensor network or domain mail servers extracting received email features. The nodes observe local information and need to share information (completely or incompletely) amongst themselves, but this requires knowing the address of other nodes for sending messages. To facilitate this, the *communication graph* layer allows the nodes to communicate by layering an overlay network on top of the underlying network topology. This has several benefits - for example, the edges in the overlay network can encode the costs and constraints of the underlying network. Furthermore, the overlays can be used for data aggregation and resiliency.

The second layer, called the *reasoning graph*, aids in designing distributed inference algorithms for reasoning about data from separate information sources. Probabilistic methods have been used to deal with situations where local observation at nodes need to be shared

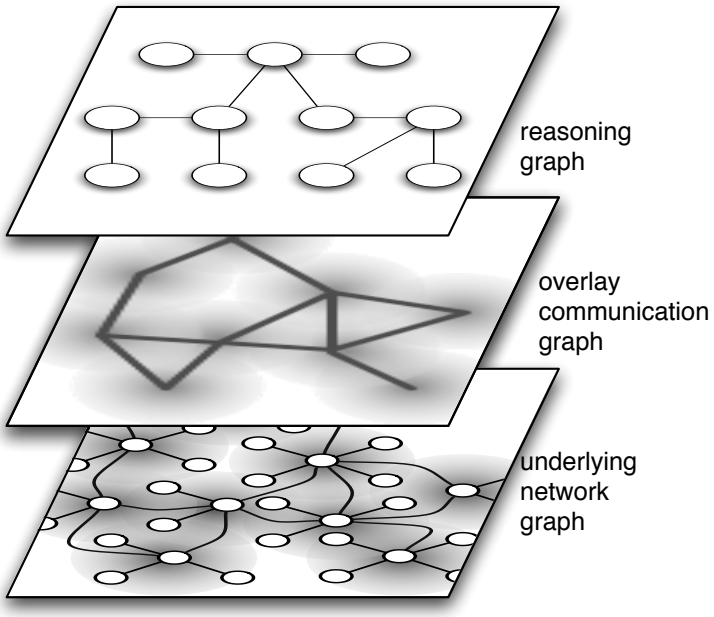


Figure 2.2. Layered graphs for distributed inference. The top two layers are implemented by a distributed inference algorithm; the lower layer is the underlying physical network.

(completely or incompletely) in order to obtain global conclusions. **Probabilistic graphical models**, a machine learning technique, use graphs to represent correlations and independencies in data which facilitates designing efficient algorithms for answering complex queries about real-world systems. Algorithms for answering these queries are also known as **inference algorithms** and can often be viewed as logical **message passing** over some graph [9]. We call this graph the *reasoning graph*.

The two graphs described above are usually not the same. For example, nodes that have highly correlated information would be connected in the reasoning graph, but they may not have a direct communication link between them in the communication graph. Moreover, some algorithms may not use a separate communication overlay graph. For example, in Section 3.1.1 junction tree inference algorithm uses a spanning tree overlay communication graph, while loopy belief propagation algorithm in Section 3.1.2 omits using a communication overlay graph.

Our approach to design distributed inference algorithms with the architecture described above (Figure 2.2), involves implementing a distributed version of existing **inference algorithms** over reasoning graphs. We use the **P2 declarative infrastructure** [24, 26] for implementing distributed versions of existing centralized algorithms over overlay communication graphs.

There have been a few distributed inference algorithms that utilize overlays. For example, Schmidt and Aberer [38] proposed using distributed hash tables (DHTs) to perform content-based addressing in loopy belief propagation. Paskin, Guestrin, and Mcfadden [31] described a robust architecture for distributed junction tree inference in sensor networks. However, each of these approaches were a point solution that were specific to the data and problem they were being used for. We propose to use the P2 declarative distributed programming framework for designing well-known inference algorithms that can be applied to a wide range of problems. We implement a declarative implementation of the distributed junction tree architecture proposed by Paskin et al. [31]. We have not implemented the scheme proposed by Schmidt et al. to use DHTs for content-based addressing and believe that this is an important application for future investigation.

2.3 Reasoning Graph

In this section, we discuss implementing distributed inference algorithms over the reasoning graph and present an application of distributed inference. Section 2.3.1 shows how an existing inference algorithm can be modeled on the reasoning graph. In Section 2.3.2 we present collaborative spam filtering as an application of distributed inference. We discuss the challenges related to distributed inference in Section 2.3.3.

2.3.1 Distributed Inference

Many probabilistic inference algorithms can be viewed as **message-passing** algorithms over the reasoning graph (Figure 2.2) and can be implemented in a distributed fashion

(for example, junction tree inference[2] [31]). In a naive distributed implementation, each variable can be a node in the network which sends messages across the network edges.

Inference algorithms like **Junction Tree Inference** provide exact solutions. Junction tree uses an overlay that has a tree structure. Unfortunately, there exist problems where junction tree inference is not feasible due to its computational complexity. For such problems **Loopy Belief Propagation**[3] [29, 46] is one of the algorithms that is used to approximate the marginals $\left\{p(y_i; x)\right\}$ instead of providing exact solutions. Loopy belief propagation is an iterative method that can be viewed as passing messages on the reasoning graph. A message from variable s to variable t is computed as

$$\mu_{s,t}(y_t) \leftarrow \sum_{y_s} \psi_s \times \psi_{s,t} \times \prod_{r \in N_G(s) \setminus t} \mu_{r,s}(y_s), \tag{2.3}$$

where ψ_s is the local potential of variable s and $\psi_{s,t}$ is the edge potential between variable s and t. $\mu_{r,s}(y_s)$ is the message s receives from neighbor r except t. The product is taken over messages of all neighbors r of variable s in the graph, other than t. At convergence, the node marginals can be approximated as

$$p(y_s) \approx \frac{1}{Z_i}\, \psi_s(y_s; x_s) \times \prod_{r \in N_G(s)} \mu_{r,s}(y_s). \tag{2.4}$$

While conceptually, we could collect all the features x to a centralized location, a distributed version of the inference algorithm (2.3) has several advantages: it eliminates single points of failure, distributes the computation across several nodes and avoids the need to send a lot of data.

In general, each network node is assigned a portion of the probabilistic model (Equation 2.3) and the nodes collaborate to compute the marginal distribution over one or more variables. In our approach we assume a networking model where each node can communicate to a subset of other nodes, but the communication costs between nodes can vary based on different cost metrics like hop-count and round-trip time.

[2]Junction tree inference is explained in detail in Section 3.1.1

[3]Loopy belief propagation is discussed further in Section 3.1.2.

2.3.2 Application

One application of distributed inference is **collaborative spam detection** [11, 5]. In collaborative spam filtering, domains wish to perform early detection of spammer IP addresses based on the emails they receive. A single domain receives only a subset of the spam from any single IP address. This hinders the domain from blacklisting the IP address since its activity can be well below the threshold for triggering an alarm at any one node [33]. Incomplete local information at a particular node raises the need for nodes to collaborate and share their information.

Domain mail servers aggregate information about senders in order to identify IP addresses that may be spammers. To classify a sender one may include several sources of information including blacklisting, local features of individual email messages and **behavioral** features [34] that align clusters of nodes with similar sending patterns. The collaborative spam filtering problem can be formulated as a graphical model where each variable y_i represents the class of the IP address (spam or ham), and it has some features x_i associated with it, like the frequency of emails sent to a set of monitoring domains. Given a set of observed features x of all domains, the marginal distribution $P(y_i; x)$ represents the likelihood that IP address i is sending spam.

2.3.3 Challenges

Having modeled an existing inference algorithm on the reasoning graph and discussed an application of distributed inference, in this section, we discuss the challenges involved in implementing distributed algorithms. In the context of distributed inference, this difficulty arises due to the following two facts: The algorithms need to use a **decentralized representation** of the probabilistic model (Equation 2.3) and perform **global coordination** to distribute the computation across several nodes. The algorithms need to be **network-aware** and should be able to perform robustly in the presence of communication delays.

A core challenge in distributed algorithms is that programming distributed systems is itself difficult. Conventionally, the high-level and compact description of these algorithms

needs to be translated manually into a set of low-level communication protocols which then needs to be executed in a low-level programming language. Such translation is often time-consuming and error-prone, and results in programs that are very difficult to debug.

Adding optimizations to distributed algorithms is also a difficult task. Simple implementations of these algorithms tend to be fairly straightforward, but for example, while effective message scheduling has been proposed in the centralized literature (to speed up the convergence of loopy belief propagation) very few papers have attempted to attain similar results in a distributed setting [37].

In this thesis, we examine how declarative languages simplify the implementation of distributed inference algorithms and customize these algorithms for optimizations.

2.4 Declarative Specification of Overlays

We use the P2 [24, 26] declarative framework in our work to implement the inference algorithms and the communication overlay network in our work. P2 takes specifications in a declarative query language, and uses database query optimization techniques to compile them into dataflow programs that resemble a mixture of traditional relational query plans and network routers.

To explain declarative networking we start with an example of specifying shortest-path routing among a set of nodes. In Figure 2.3 we present this protocol in the traditional recursive query language *Datalog* enhanced with aggregation functions [16]. Datalog programs consist of a set of declarative *rules*, terminated by periods. The right-hand-side of the rule represents a conjunctive predicate over relations in a database, and the left-hand side represents the deduction from that predicate. For example, rule **D1** can be read as "**if** there is a tuple `(Src, Root)` in the `intree` relation, **and** there is a tuple `(Src, Root, Cost)` in the `link` relation (where the `Src` and `Root` variables of the two tuples match), **then** there is a tuple `(Src, Root, Root, Cost)` in the `path` relation". This rule identifies paths to the root of each tree from that root's children; those paths have the cost of the corresponding

```
/* if there is a tuple (Src, Root) in the intree relation, and a tuple (Src, Root, Cost)
in the link relation, then there is a tuple (Src, Root, Root, Cost) in the path relation
which states that Src can reach Root through Root */
D1:   path(Src,Root,Root,Cost) :-
                  intree(Src,Root),
                  #link(Src,Root,Cost).

/* if Src has a link to NextHop node which has a path to the Root, then the Src has a
path to the Root via NextHop */
D2:   path(Src,Root,Nexthop,Cost) :-
                  intree(Src,Root),
                  #link(Src,NextHop,C1),
                  path(NextHop,Root,Hop2,C2),
                  Cost = C1 + C2.

/* From all the learnt paths to the Root find the minimum cost path and set the Nexthop of
the minimum cost path as the parent */
D3:   minCost(Src,Dst,min<Cost>) :-
                  path(Src,Dst,NextHop,Cost).
D4:   parent(Src,Root,NextHop,Cost) :-
                  minCost(Src,Root,Cost),
                  path(Src,Root,NextHop,Cost).
Query:   parent(Src,Root,NextHop,Cost).
```

Figure 2.3. Shortest-Path Routing in Datalog.

one-hop link from child to root, and the path's "next hop" for routing (the third field of the `path` relation) is, the root itself. This is the base case of a recursive path-finding specification. The recursive case is captured in rule **D2**: **if** a source node has a link to another "NextHop" node, **and** that node in turn has a path to the tree's root, **then** the `src` node has a path to the root *via* that "NextHop" node, with the appropriate cost.

These two rules are sufficient to find all possible source–root paths. The next two rules prune this set: **D3** uses the aggregation function `min` to identify the cost of the shortest (least-cost) path from each source to each root. **D4** sets `parent` of each source in each tree

```
ND1:  path(@Src,Root,Root,Cost) :-
                intree(@Src, Root),
                link(@Src,Root,Cost).
ND2:  path(@Src,Root,Nexthop,Cost) :-
                intree(@Src, Root),
                link(@Src,NextHop,C1),
                path(@NextHop,Root,Hop2,C2),
                Cost = C1 + C2.
ND3:  minCost(@Src,Dst,min<Cost>) :-
                path(@Src,Dst,NextHop,Cost).
ND4:          parent(@Src,Root,NextHop,Cost) :-
                minCost(@Src,Root,Cost), path(@Src,Root,NextHop,Cost).
Query:  parent(@Src,Root,NextHop,Cost).
```

Figure 2.4. Shortest-Path Routing in Overlog (Network Datalog).

to be the `NextHop` in the shortest path. Finally, the query specifies that all such `parent`s should be returned as output.

The Datalog program of Figure 2.3 is sufficient to specify shortest paths routing between a set of nodes. However, this requires a network protocol since Datalog assumes that the relevant data, and the query processing computation, are centralized on a single computer. Figure 2.4 shows a variant of the program expressed in the *Overlog* language, which specifies a workable, distributed network protocol based on that logic. The Overlog code in Figure 2.4 has one field prepended with the "`@`" symbol; this field is called the *location specifier* of the relation. The location specifier specifies physical data distribution: each tuple is to be stored at the address in its location specifier field. For example, the `path` relation is partitioned by the first (source) field; each partition corresponds to the networking notion of a local routing table.

The *link* relations have names that contain two fields of type `address` representing source and destination nodes in the network. These predicates capture edges in the underlying network graph and correspond to the networking concept of a local neighbor table.

Loo et al. [26] show how location specifiers and link relations enable an Overlog compiler to generate a distributed protocol guaranteed to be executable over the underlying network topology captured by the link relation.

2.5 Summary

In this chapter, we gave an overview of the layered graph architecture for implementing distributed inference algorithms and presented how to model a distributed version of an existing inference algorithm using this architecture. We then discussed an application of distributed inference and also discussed the challenges in designing distributed algorithms. We also gave an overview of the P2 declarative language, Overlog. In the next chapter, we will present declarative distributed implementation of existing inference algorithms like junction tree inference and loopy belief propagation.

Chapter 3

Declarative Implementation of Existing Inference Algorithms

Having given an overview of the declarative programming language Overlog, and the architecture to implement distributed inference algorithms, this chapter presents the distributed implementation of existing inference algorithms like junction tree inference and loopy belief propagation. We show in this chapter that declarative programming aids in concise and easy implementation of these algorithms. Moreover, we could easily customize our Overlog implementation of loopy belief propagation to obtain faster convergence.

Section 3.1 explains junction tree inference and loopy belief propagation, and their declarative distributed implementation. We evaluate Overlog implementations of junction tree inference and loopy belief propagation, and compare them with existing distributed implementations in Section 3.2.

3.1 Inference Algorithms

Many existing inference algorithms can be viewed as message passing algorithms over the reasoning graph. For example, the junction tree algorithm [9] uses a reasoning graph that takes the form of a *tree*. The nodes in the tree pass local information in terms of

messages and provide exact solution. Computational complexity of junction tree inference grows exponentially with the size of the maximal clique in the tree. We discuss junction tree inference algorithm in detail in Section 3.1.1.

Many practical problems that do not have large maximal cliques can be modeled by the junction tree algorithm and solved efficiently. In a distributed setting nodes at multiple locations can connect in the form of a tree and pass messages. For trees that have large cliques, instead of using methods that provide exact solutions approximate methods are preferred. These methods also have interesting reasoning graphs. For example, loopy belief propagation [32, 10] is often used for graphs with loops. This algorithm does not give an exact answer, but can provide reasonable solutions in practice. Section 3.1.2 discusses the declarative implementation of loopy belief propagation.

3.1.1 Junction Tree Inference

As mentioned before, the junction tree inference algorithm is used to provide exact solutions and uses a reasoning graph takes the form of a tree. Each node in the tree corresponds to a *set (or clique) of random variables*. To guarantee the correctness of the message passing algorithm on the junction tree, the cliques must satisfy a structural constraint called the **running intersection property:** Specifically, for each pair of nodes m, n:

$$X \in C_m, X \in C_n \implies X \in C_k \tag{3.1}$$

for all nodes k on the (unique) path between m and n. Intuitively, since both clique m and n have information about X then all nodes in between the two nodes along the path in the tree must have observed information about X. A tree that satisfies this property is called a **junction tree**. The complexity of the junction tree message passing algorithm grows exponentially with the size of the maximal clique.

Distributed junction tree inference has been used by Paskin et al. [31] in sensor networks for sensor calibration. The architecture proposed by Paskin et al. requires two graphs: spanning tree and junction tree. We can easily model the reasoning graph as a spanning tree, where each node is associated with a random variable. Junction tree is layered on

top of the reasoning graph. Junction tree satisfies the running intersection property by performing clique calculations based on the neighbor relationship between nodes in the spanning tree. Inference is performed by sending messages between neighboring nodes to find exact solutions.

We examine the **distributed junction tree** problem in [31] and show that the architecture in the paper can be naturally expressed in a declarative framework. The architecture [31] computes the running intersection property as follows: Each node n begins with a set of **local** variables L_n. The network nodes form a **network junction tree**, that is, a spanning tree over the network communication graph such that each node is associated with a clique $C_n \supseteq L_n$, and the cliques $\{C_n\}$ satisfy the running intersection property (3.1).

The distributed algorithm of Paskin et al. [31] consists of three layers:

1. **Spanning tree formation**: The nodes form a spanning tree. The spanning tree responds to changes in network connectivity and node failures.

2. **Junction tree formation**: The nodes compute a set of minimal cliques $\{C_n\}$ that satisfy the running intersection property (3.1).

3. **Inference**: The inference layer allows nodes to transfer messages between themselves to calculate the probabilistic inference at a particular node. For example, in probabilistic inference, each node starts with a potential ψ_{C_i}[1] over (a subset of) its clique C_i. Each node i then computes the inference message μ_{ij} to its neighbor node j based on the messages received from its other neighbors k.

$$\mu_{ij} = \sum_{C_i \setminus S_{ij}} \psi_{C_i} \prod_{k \neq i} \mu_{ki} \tag{3.2}$$

Once the node i has received messages from all of its neighbors, we compute the marginal probability update for i:

$$p(C_i) \propto \psi_{C_i} \prod_{k} \mu_{ki} \tag{3.3}$$

[1] A potential is a non-negative function associated with each clique (Section 2.1)

These layers are executed in parallel: for example, if an edge in the spanning tree is added or removed, the junction tree layer updates the clique at each node and the inference layer recomputes the inference messages.

In the original implementation of Paskin et al. [31], the three layers had to explicitly handle changes in the underlying network, listening to local events indicating the change such as edge addition and deletion. Such complex interleaving of the inference and networking layers can be naturally expressed in a declarative language. For example, to implement the *running intersection property* the clique at node n of the network junction tree is computed as

$$C_n = L_n \bigcup \left\{ i : i \in R_{n,m} \cap R_{n,k}, m \neq k \right\}, \tag{3.4}$$

where $R_{n,m}$ is the set of all variables in the subtree rooted at node n, with leaf as node m and L_n are the local variables at node n. This equation directly maps to rule C1 and C2 in the declarative implementation in Figure 3.1 [2]. The reachable variables relation $R_{m,n}$ is naturally expressed recursively with R1 (the base case) and R2 (the recursive case). The P2 runtime responds to any changes to the preconditions, and recomputes the reachable sets and cliques as necessary.

In a distributed environment we cannot assume that the physical nodes are stable or that we can know when the communication graph has stabilized. There can be node failures as well as additions which cause the spanning tree to detect the event and update the tree to remove or add the node. Additions or removals of nodes cause updates to the junction tree cliques. This requires the need to have the three layers run simultaneously in order to respond to changes between the layers.

Our Overlog implementation of the junction tree inference has separate modules for the three layers. To give a feel of the declarative rules and their conciseness we briefly explain the important rules of these layers:

- **Spanning tree rules:** Our distributed spanning tree algorithm is similar to that

[2]The declarative specification of running intersection property in Figure 3.1 does not specify the @ location specifiers. It lists the pseudo datalog code for the algorithm. Refer to Figure A.1 in Appendix A for the Overlog specification of running intersection property

```
/* if node n has a local variable i, then neighboring node m can reach variable i through
node n.  This information is stored in reachable relation */
R1:   reachable(m,n,i) :-
                  localvar(n,i),
                  neighbor(n,m).

/* if node n can reach variable i through neighboring node k, then neighbor node m (m
!= k) can reach variable i through node n.  */
R2:   reachable(m,n,i) :-
                  reachable(n,k,i),
                  neighbor(n,m),
                  m != k.

/* if variable i can be reached through neighbors m and k, then the clique at node n
should have variable i*/
C1:   clique(n,i) :-
                  reachable(n,m,i),
                  reachable(n,k,i),
                  m != k.

C2:   clique(n,i) :-
                  localvar(n,i).
```

Figure 3.1. Implementing distributed triangulation in Overlog. The statement reachable(n,k,i) represents the fact that the variable i is in the subtree (spanning tree) rooted at node n, with leaf k.

of Paskin et al. [31]. We give here a brief overview of the spanning tree Overlog that contains 20 rules. Refer to Appendix A.3 and A.4 for the complete Overlog specification. For spanning tree root election, each node is assigned an identifier and the node with the smallest identifier becomes the root of the spanning tree. Two important base relations at each node are *pulse* and *config*. To coordinate with other nodes, rule *c1* and *c2* periodically broadcast a configuration message from each

node to its neighbors, which conveys each node's current choice of root and parent. The configuration messages received by a node are stored in the *config* relation. Periodically each node executes rule *r1*, *r2*, *r3*, *r4* and *r5* to select a parent that has the lowest root amongst all potential parents learnt through the tuples in the *config* relation; this ensures that all nodes agree on the root. In order to ensure that stale configuration messages are not considered while determining the parent of a node, each node adds pulse information in the configuration messages. The *pulse* relation at each node is updated through rules *p1*, *p2*, *p3* and *p4* to store the latest pulse heard from a node.

- **Junction tree rules:** Our junction tree Overlog implements the running intersection property and contains 7 rules. Refer to Appendix A.1 for the complete Overlog of running intersection property. Each node has a partition of the *localVars* base relation that stores the local variables associated with each node. The *reachVars* base relation stores information regarding the *reachable* variables associated with the subtrees of each node. The equation for the running intersection property (Equation 3.4) directly maps to rule *C*. The reachable variables relation $R_{m,n}$ is naturally expressed recursively with rule *R1* (the base case) and rule *R2* (the recursive case). The *clique* relation represents the clique associated with each node. At each node it stores its local variables and other variables that are added from rule *C*. This is the clique associated with each node in the junction tree. The *separator* relation stores the common variables between each node and each of its neighbors.

- **Inference rules:** The inference Overlog contains 10 rules. Refer to Appendix A.2 for the complete specification. Periodically message updates are fired by rule *m1*. The *incoming* relation stores all the messages a node receives from its neighbors, μ_{ki} in Equation 3.2 and local potential ψ_{C_i} associated with clique at each node is stored in *localFactor* relation. Rule *m5* takes the list of all potentials and generates μ_{ij} message that is sent from each node i to its neighbor j. Functions *f_product* and *f_marginal* encapsulate the operations to be performed for generating messages μ_{ij}

in Equation 3.2. Belief calculations in rule *m8*, *m9* and *m10* calculate the marginal probability in Equation 3.3.

We provide an evaluation of the declarative specification of the junction tree algorithm with respect to the Lisp implementation of Paskin et al. [31] in Section 3.2.1.

3.1.2 Loopy Belief Propagation

The computational complexity of junction tree makes it infeasible for graphs with large cliques. As mentioned in Section 3.1, in such situations loopy belief propagation [32, 10] is often used to provide approximate solutions. Intuitively, the algorithm only requires nodes to agree locally about neighboring variables in the graph, and since the graph may have loops this local consistency does not lead to a globally consistent solution. In a junction tree, there is only one path between nodes (since it is a tree), but the nodes must agree on entire cliques of variables.

Recall that loopy belief propagation is an iterative method and can be viewed as passing messages on the reasoning graph. A message from variable s to variable t is computed as

$$\mu_{s,t}(y_t) \leftarrow \sum_{y_s} \psi_s \times \psi_{s,t} \times \prod_{r \in N_G(s) \setminus t} \mu_{r,s}(y_s), \tag{3.5}$$

where ψ_s is the local potential of variable s and $\psi_{s,t}$ is the edge potential between variable s and t. $\mu_{r,s}(y_s)$ is the message s receives from neighbor r except t. The product is taken over messages of all neighbors r of variable s in the graph, other than t. At convergence, the node marginals can be approximated as

$$p(y_s) \approx \frac{1}{Z_i} \psi_s(y_s; x_s) \times \prod_{r \in N_G(s)} \mu_{r,s}(y_s). \tag{3.6}$$

All nodes calculate outgoing messages based on the incoming messages from their neighbors. In a centralized implementation, this is performed iteratively at each node in a synchronous fashion. The messages are said to converge if none of the messages in the later successive iterations cause any changes or the changes are below a particular threshold.

The distributed implementation of the above naive loopy belief propagation scheme has two problems. First, all nodes need to coordinate to have synchronous execution of each iteration of updates. Achieving synchronization is difficult in a distributed implementation. Second, the algorithm sends all messages to the neighbors without focusing on messages that aid in faster convergence. In a distributed setting, where bandwidth and power consumption are often the limiting factors, updating the messages indiscriminately can be especially costly. A paper by Elidan et al. [13] proposed an effective centralized solution that updates messages in the order given by a lower-bound on the difference between the current message and the previous computed message. This scheme requires implementing a global priority queue which is difficult to implement as well as manage in a distributed setting. In this section, we describe a simple randomized approximation of this algorithm. Our algorithm is similar in spirit to [37], but uses a spanning tree to compute an implicit normalization constant in the algorithm.

A simple strategy, proposed by [37] is to delay messages with smaller residuals. This strategy can be implemented by performing a sequence of independent Bernoulli trials. At each iteration, the message $\mu'_{s,t}$ is transmitted with probability given by

$$p_{s,t} = \left\| \mu'_{s,t} - \mu_{s,t} \right\|^{\rho} \tag{3.7}$$

for a suitably chosen constant $\rho \geq 0$ (here, $\mu_{s,t}$ is the last transmitted message). Effectively, the messages with larger residuals $r_{s,t} = \|\mu'_{s,t} - \mu_{s,t}\|$ will be less likely to wait for transmission.[3]

The algorithm, as described so far, has a drawback. As the iterations are performed and the messages get closer to the fixed point, the transmission probabilities $\{p_{s,t}\}$ decrease throughout the network. Therefore, the algorithm will eventually stop making progress and will *never* converge. In order to ensure convergence, we need to multiply the update probability in Equation 3.7 with a suitably chosen normalization term λ. In a distributed setting, we can use a spanning tree to compute the sum [27] of the norms (3.7), and normalize the probabilities $\{p_{s,t}\}$ to some pre-determined update rate u. The normalization

[3]Here, it is assumed that the residual $r_{s,t} \leq 1$.

constant is computed as:

$$N = (\sum_{s,t} r^{\rho}_{s,t})/u \tag{3.8}$$

N is computed periodically, which ensures that the algorithm continues to make progress. As we demonstrate in Section 3.2.2, the resulting **randomized loopy belief propagation** algorithm offers substantial improvement over the naive synchronous iterative algorithm.

Our Overlog implementation of the above proposed randomized scheme reuses the spanning tree implementation of junction tree inference discussed in Section 3.1.1. We use spanning tree to aggregate the sum of the norms and share them to calculate the normalization constant in Equation 3.8. The Overlog specification of normalization calculation is listed in Appendix B.5. Rule N updates the constant based on the received residuals. Overlog rules for message, residual and belief calculations are similar to that of naive loopy belief propagation. We modify rule *l1* and add two predicates that read normalization constant from the *normalizer* table and include the logic of Equation 3.7 by using function *f_coinFlip* that performs Bernoulli trails. For the complete Overlog specification refer Appendix B.3, B.4.

3.2 Evaluation

In this section, we evaluate the distributed declarative implementation of the junction tree inference and loopy belief propagation. All experiments have been performed on the Emulab testbed [47].

The first set of experiments evaluate our declarative junction tree inference algorithm. The goal of these experiments is threefold. First, we aim to compare the conciseness of our declarative distributed implementation with the distributed Lisp implementation of Paskin et al. [31]. Second, we examine the bandwidth consumption of our implementation in case of network partitions. Third, we compare our implementation with that of Paskin et al.'s Lisp implementation in terms of message complexity. We do not expect our overall

performance to be as good as the hand-coded Lisp implementation, but we would like to show that our implementation can be used for sensor calibration in sensor networks.

Our second set of experiments evaluate randomized loopy belief propagation algorithm proposed in Section 3.1.2. Our goal for these experiments is threefold. First, we want to verify whether randomized belief propagation converges much faster compared to naive loopy belief propagation. Second, we want to illustrate that the convergence rate of our randomized loopy belief propagation algorithm (Section 3.1.2) is comparable to the centralized residual belief propagation algorithm [13]. Third, we evaluate the bandwidth of different components of the randomized scheme and see the tradeoff between increase in bandwidth and faster convergence.

3.2.1 Junction Tree Inference

We use the sensor calibration dataset used by Paskin et al. [31] to evaluate our declarative junction tree inference algorithm with the hand-coded Lisp implementation of Paskin et al. [31]. In our Overlog implementation, the link quality information is provided externally rather than being estimated explicitly from the configuration messages.

In our first experiment we compare the conciseness of our declarative junction tree implementation with the Lisp implementation. Table 3.1 illustrates that we achieve roughly 4 times reduction in terms of lines of code.

Protocol	Lisp	Overlog
Spanning Tree	848 lines (9452)	274 lines (2589)
Triangulation	457 lines (5812)	105 lines (1092)
Inference	574 lines (6040)	131 lines (1144)

Table 3.1. Number of lines and the program size (in gzip-bytes) of different modules in Overlog and Lisp.

Our second experiment evaluates our implementation of junction tree inference in terms of bandwidth consumed in maintaining the three layers: spanning tree, junction tree (running intersection property) and inference. We emulate network dynamics by

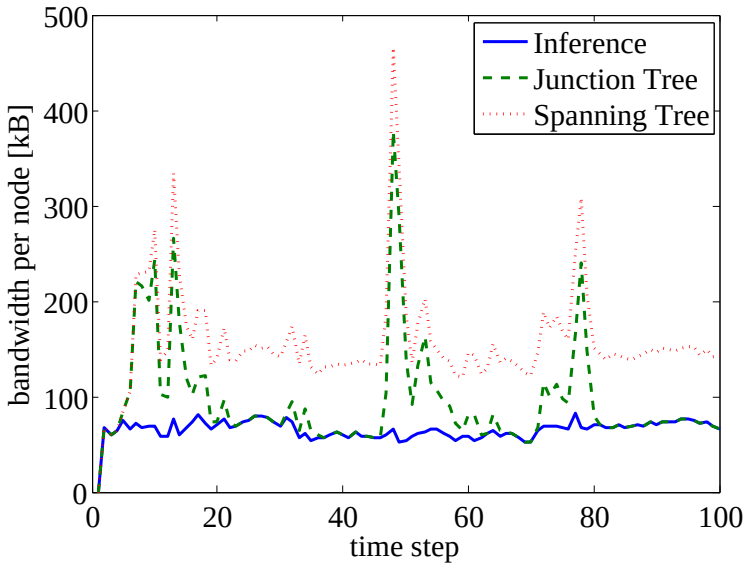


Figure 3.2. Distributed Junction Tree Inference evaluation: An example run for a network with 54 nodes. Partition occurs at time 45, and the communication is restored at time 70. Each bandwidth curve is a cumulative of the curves plotted under it.

partitioning the graph at time-step 45 and then re-connecting the partitions at time-step 70. Partitioning of the network is performed by emulating dead links.

Figure 3.2 shows that our implementation is robust and recovers from failures involving extreme conditions like network partitioning. The bandwidth usage peaks up to 475 KBps in case of network partitioning. Soon after links go down or come up, communication increases as messages are sent to recalculate the spanning tree and restore the running intersection property. Note that the communication cost of repairing a partitioned network is higher compared to building the initial tree or joining a partitioned network.

In Figure 3.3, we evaluate the message complexity of different components of the architecture in [31], in the P2 and Lisp implementation respectively. Since the two codebases use a different serialization mechanism, we show the results as a function of the total number of tuples transmitted. We can see that the spanning tree message counts of both implementations are comparable. The reason why our declarative implementation of

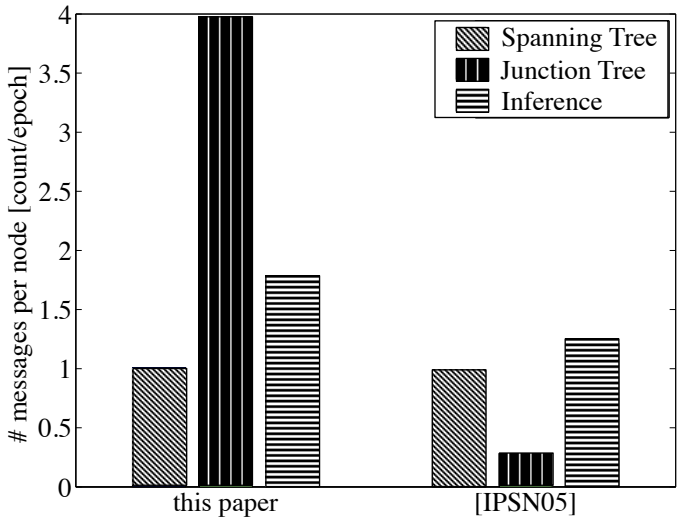


Figure 3.3. Distributed Junction Tree Inference Message Count Comparison: The communication requirements of two implementations of the architecture in [31].

junction tree and inference has higher message count is because the Lisp implementation performs periodic updates to propagate the changes while our Overlog implementation propagates changes as soon as they happen. Periodic updates reduce the message count because multiple changes between each epoch are combined and optimized. Message complexity of junction tree and inference can be reduced by adopting the periodic update model.

Instead of implementing the periodic update model we propagate all changes as they happen because it was more easy to implement in our declarative language, Overlog. Since our main goal was to see whether Overlog provides a concise implementation of junction tree inference, we did not update the implementation to incorporate periodic updates. Although, we feel that periodic update scheme is an important aspect of the application and should be implemented to assess the tradeoff between code size and performance.

3.2.2 Loopy Belief Propagation

In this section, we evaluate our proposed randomized loopy belief propagation scheme with respect to other variations of loopy belief propagation discussed in Section 3.1.2.

In order to evaluate different variations of loopy belief propagation we begin by considering random grids, parameterized by the ising model[4]. A random grid with NxN binary variables is generated. An ising model is used for each edge potential where an edge has the potential $f(x_i, x_j) = \exp(\beta_{i,j})$ when $x_i = x_j$, and $f(x_i, x_j) = \exp(-\beta_{i,j})$ when $x_i \neq x_j$. We sampled the $\beta_{i,j}$ uniformly in the range $[-C; C]$ with $C = 5$. Similar models have been employed for testing loopy belief propagation algorithms in the past [13].

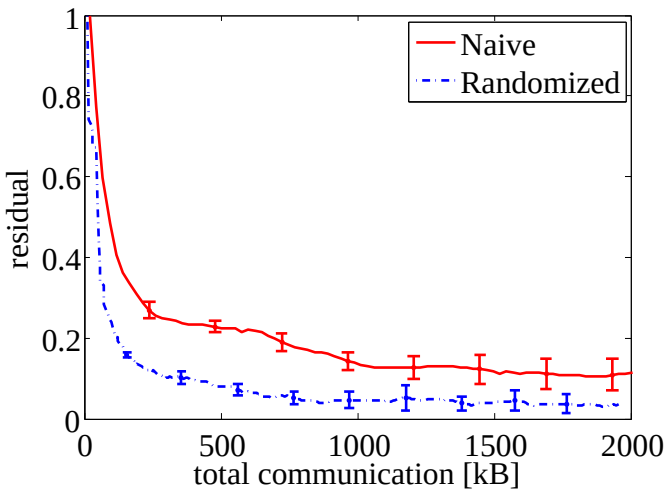


Figure 3.4. The convergence of two distributed loopy BP algorithms as a function of the bandwidth used. Both algorithms are run for a fixed number of iterations.

In our first experiment we show the comparison of the convergence of our randomized algorithm with the distributed naive iterative loopy belief propagation. In naive loopy belief propagation, we set the epoch of each iteration such that messages from all nodes have been received by their neighbors. This is done to achieve parallel synchronized iteration at each node. We evaluate the convergence in terms of the residual between the

[4]Ising model has two properties: First, each vertex is assigned one of the two states (+1 or -1). Second, each edge has an assigned constant usually written as J_{ij}, where i and j are the two vertices.

current and the old message received at a node. As we can see in Figure 3.4, randomized belief propagation converges faster than the iterative algorithm that propagates all messages equally. This verifies our intuition of achieving faster convergence and lower communication cost by transmitting messages with larger residuals and normalizing the probability of transmission to ensure convergence.

In our second experiment, we evaluate the convergence of our randomized belief propagation algorithm with different ρ values (ρ represents the convergence factor in Equation 3.7) and compare it with centralized residual belief propagation algorithm [13]. Figure 3.5 shows the residual as a function of the number of updates performed by the algorithm. As seen in figure 3.5, our randomized scheme performs nearly as well as the centralized residual algorithm by Elidan et al., in terms of convergence. We see that the distributed algorithms converge faster as ρ is increased. This is because as ρ increases the messages with high residuals are more likely to come first while messages with low residuals are ignored. Thus, as ρ increases the algorithm behaves similarly as the centralized residual belief propagation algorithm [13].

Figure 3.6 shows bandwidth requirements of different components of randomized belief propagation compared to the synchronous belief propagation. We see that the global coordination of the normalization constant (Equation 3.8) (achieved by spanning tree and aggregation), required by the randomized scheme, increases the communication complexity of the solution only mildly. Synchronous belief propagation has only the inference component and this is the reason why the total communication cost of the algorithm is equal to the inference component.

3.3 Summary

In this chapter, we presented our distributed implementation of existing inference algorithms using the declarative framework. In summary, we made the following observations from our evaluation results:

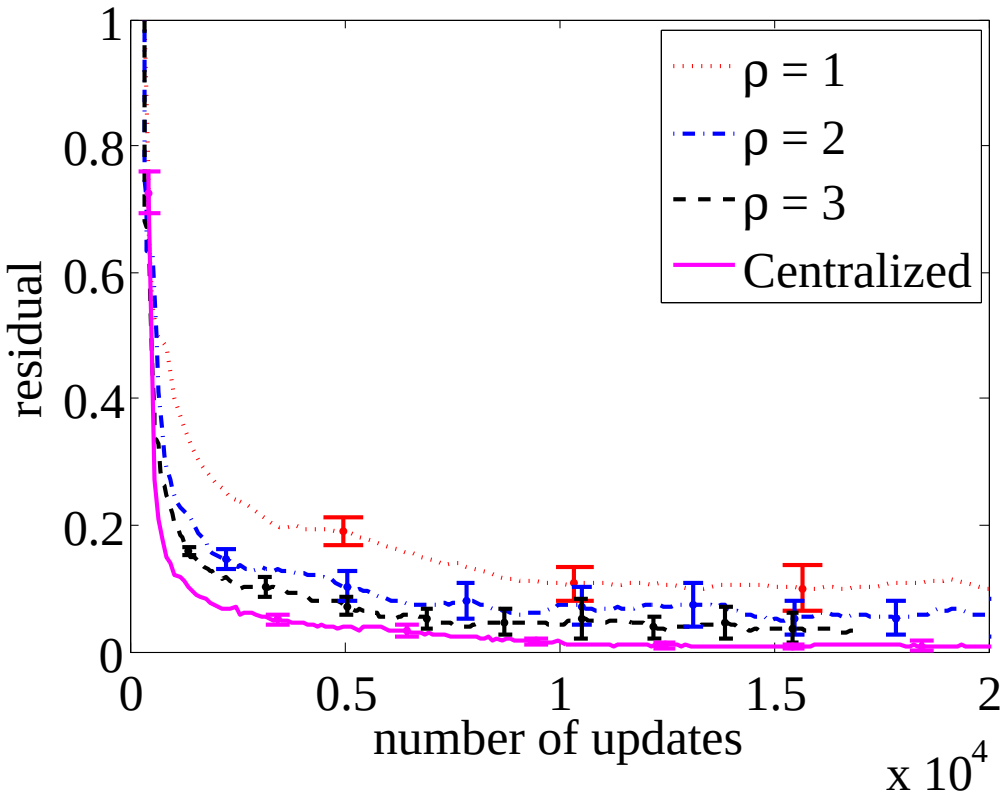


Figure 3.5. The effect of ρ parameter in the randomized belief propagation algorithm. As ρ increases, the algorithm converges to the centralized residual belief propagation algorithm.

- Our declarative implementation of the existing junction tree algorithm is concise and easy to implement. Our implementation has 4 times less code compared to the existing distributed Lisp implementation of junction tree by Paskin et al. [31].

- The Overlog declarative program for junction tree inference robustly handles extreme conditions like network partitioning. All three layers: spanning tree, junction tree and inference, run in parallel and get updated as links go down and come up.

- The declarative programming language, Overlog, helped us in customizing existing algorithms and reusing code. We could easily customize the naive iterative loopy belief propagation algorithm to add a randomized message scheduling scheme, to achieve faster convergence and reduction in consumed bandwidth. Moreover, we

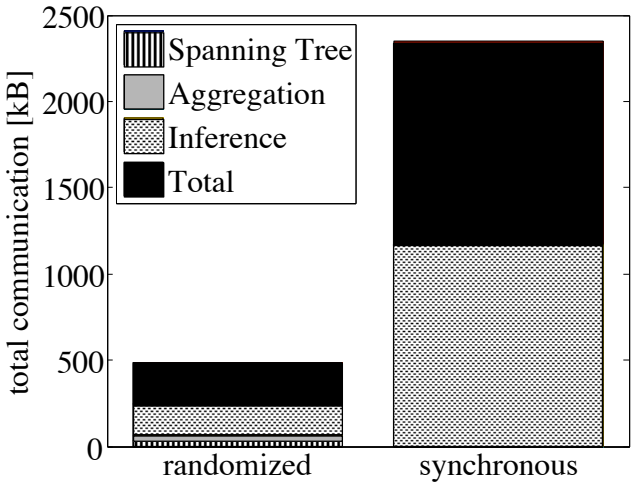


Figure 3.6. The bandwidth requirements of different components of our randomized algorithm compared to naive synchronous loopy belief propagation algorithm. Both algorithms are run until convergence.

could reuse the declarative implementation of spanning tree (used in the junction tree inference) for aggregation to estimate the normalization constant.

- Global coordination of the normalization constant (achieved by spanning tree and aggregation), required by the randomized scheme, increased the communication complexity of the solution only mildly.

The observations discussed above show that declarative programming framework accelerates the implementation of distributed inference algorithms by providing conciseness of code, concentrating on the high-level concept of *what* has to be implemented than worrying about low-level implementation, code reusability and ease of customization.

In the next chapter, we apply our declarative distributed inference architecture to a network monitoring application for spam detection.

Chapter 4

Collaborative Spam Filtering

Having shown that distributed implementation of existing inference algorithms can be easily implemented using the P2 declarative programming framework, we now explore applications of distributed inference in the context of network monitoring. We take a centralized and offline scheme of spam detection from the work of Feamster et al. [34] and see whether declarative programming makes it easy to build a distributed implementation of the same.

This chapter is organized as follows. In Section 4.1, we discuss the need for collaborative spam filtering. Section 4.2 explains the clustering and classification scheme used by the spam filtering application, along with the clustering algorithm chosen for our distributed implementation. We explain the architecture of our distributed implementation in Section 4.3. We evaluate our distributed implementation with the Feamster et al. implementation [34] in Section 4.4. In the course of designing the system, we faced issues related to choice of applicability of clustering algorithms, partitioning of data, aggregation and scalability. We discuss these issues in Section 4.5 and propose how they may be addressed. Section 4.6 discusses related work in the area of spam detection.

4.1 Introduction

Today, a large fraction of spam comes from botnets, large groups of compromised machines controlled by a single entity. Spammers distribut the job of sending spam across the compromised machines. Detecting such coordinated attacks require the ability to combine observed data collected from multiple vantage points. This raises the need of **collaborative spam filtering**, where multiple entities, which can be ISPs, domains or enterprises, to collaborate in order to share their observations with each other.

We take the **collaborative spam filtering** [11, 5] application as a first step towards using distributed inference for network monitoring problems. In collaborative spam filtering, email servers aggregate received email information in order to identify machines that are spammers. To classify a machine, one may include several sources of information including blacklisting, local features of individual email messages and **behavioral** features [34] that cluster nodes with similar sending patterns.

In collaborative spam filtering, domains wish to perform early detection of spammer IP addresses based on the emails they receive. A single domain receives only a subset of the spam from any single IP address. This hinders the domain from blacklisting the IP address since its activity can be well below the threshold for triggering an alarm at any one node [33]. Incomplete local information at a particular node raises the need for nodes to collaborate and share their information.

A key challenge in collaborative spam filtering involves spammers changing their IP addresses, which make blacklisting on the basis of IP address ineffective [33]. Recently, Ramachandran et al. [34] proposed a behavioral blacklisting technique that classifies email senders based solely on their sending behavior. For each sending IP address, the method computes the frequency of emails sent from the IP address to a set of recipient domains. They apply spectral clustering to identify clusters of IP addresses with a similar pattern of targeted domains. They find that benign senders have diverse sending patterns and unlike spammers do not form large clusters.

The SpamTracker system developed in [34] is centralized, and our aim here is to develop

a distributed version of this system. Conveniently, clustering of sender IP addresses can very easily be implemented by a distributed inference algorithm. Currently, our work leverages the declarative programming environment provided by P2 to implement the distributed implementation of SpamTracker [34]. Our implementation uses the **affinity propagation** [14] clustering algorithm that is a message passing algorithm [9] and can be implemented on a reasoning graph (Figure 2.2).

In the course of designing the system we faced issues related to choice of applicability of clustering algorithms, partitioning of data, aggregation and scalability. We discuss these issues in Section 4.5 and propose how they may be addressed. As a result of these issues, we could not run large experiments even though our preliminary experimental results indicate that our distributed affinity propagation clustering method gives similar results to the centralized spectral clustering method used by SpamTracker [34].

4.2 Clustering and Classification

In this section we explain two phases involved in the SpamTracker implementation: clustering and classification. We discuss clustering in Section 4.2.1 and classification in Section 4.2.2. These phases are similar to the SpamTracker implementation [34], except that we use a different clustering algorithm called affinity propagation. We give a brief overview of affinity propagation in Section 4.2.3.

4.2.1 Clustering

In SpamTracker [34] the clustering algorithm is used for clustering sender IP addresses based on their sending pattern, and obtaining the sending pattern of spammers. We use a different clustering algorithm than used by SpamTracker. Instead of using spectral clustering, we use **affinity propagation** [14] for clustering IP addresses.

Affinity propagation by Frey et al. [14] is a message passing algorithm that clusters similar data points. The algorithm applies the max-product algorithm (a variant of the

sum-product algorithm used in belief propagation) on a graph, and can be easily modeled on the reasoning graph. It computes a set of exemplars that represent the center of each cluster.

The clustering algorithm is used to cluster spammers that have similar sending behavior, where the behavior is determined by finding the domains a spammer targets. Email contents are not taken into consideration. To gather data concerning a spammer's sending behavior, we collect data from over two hundred distinct email domains. In the clustering phase, we generate a matrix N nxd, where n is the number of IP addresses that sent email to any of d domains within any of t particular time windows. $M(i, j, k)$ denotes the number of times IP address i sent email to domain j in time slot k.

$$N(i,j) \leftarrow \sum_{k=1}^{t} M(i,j,k) \tag{4.1}$$

In order to get the similarity matrix S nxn, which is used in the affinity propagation algorithm, the dot product between two IP addresses is calculated. When calculating the similarity that data point i shares with data point j, the dot product is normalized.

$$S(i,j) \leftarrow \frac{N(i,k) \bullet N(j,k)'}{\sum_{k=1}^{d} N(j,k)} \tag{4.2}$$

The clusters generated in the clustering phase do not have common IP addresses between them. Each cluster represents a spammer traffic pattern. The cluster center is computed by averaging the traffic pattern of the IP addresses present in the cluster. The cluster average is a 1xd vector.

$$c_{avg}(j) \leftarrow \frac{\sum_{i=1}^{|C|} N_c(i,j)}{|C|} \tag{4.3}$$

4.2.2 Classification

In the classification phase, the sending pattern of an IP address T, a 1xd vector, is determined, and then a score is calculated to determine the similarity of its traffic pattern with one of the clusters. This score is the maximum of the normalized dot product between

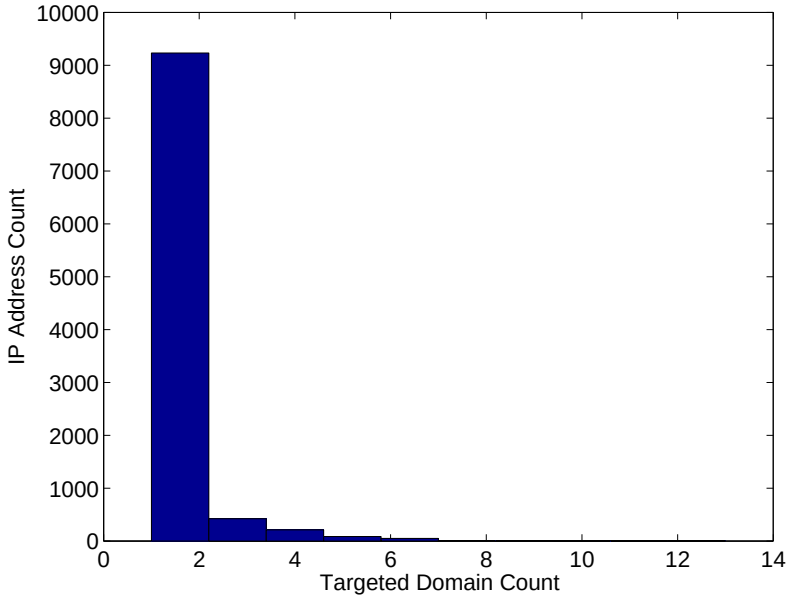


Figure 4.1. Count of domains that a benign IP address sent email to in a duration of 6 hours

the sending pattern of the IP address being tested and the set of cluster averages found in Equation 4.3.

$$Score \leftarrow \max_{1 \leq i \leq |C|} \frac{T(1,j) \bullet c_{avg}(i,j)'}{\sum_{j=1}^{d} c_{avg}(i,j)'} \tag{4.4}$$

Feamster et al. suggest in SpamTracker [34] that, since we are attempting to determine the sending patterns of spammers across multiple domains, we should ignore clusters that are dominated by very few domains. Also, spammers that target few domains will give high scores to benign senders since benign senders tend to send emails to small number of domains in a short duration (Figure 4.1). We take the same approach and ignore clusters that are spread across very few domains.

A single round of classification involves classifying the set of IP addresses that send email within a period of six hour based on the clusters found in the preceding six hour time interval.

4.2.3 Affinity Propagation

Affinity propagation is a message passing algorithm that clusters similar data points. The algorithm takes as input a similarity matrix S of nxn size for n data points (Equation 4.2). $S(i, j)$ represents the similarity data point i has with data point j. The similarity matrix does not need to be symmetric, that is the similarity data point j has with data point i does not need to be the same as data point i's similarity with data point j. Data point k's similarity with itself is referred to as its preference.

Preference of data point k represents a priori suitability of it being an exemplar. In affinity propagation preference values play an important role in determining the number of clusters generated. Thus, instead of setting all preferences to a value of one, all preference values are set to a common value. Good initial choices include setting preference of all data points to the minimum or median similarity. For our experiments, the preferences were set to the median similarity of a given IP address.

There are two kind of messages that are sent between data points: *Availability messages a(i,k)* are sent from candidate exemplars k (data points) to other data points i, indicating how appropriate that candidate would be as an exemplar and *Responsibility messages r(i,k)* are sent from data points k to candidate exemplars i (data points), signaling how well-suited the data point k is to serve as the exemplar for point i.

Both these messages are updated iteratively based on each others value. First iteration starts with initializing *availability messages a(i,k) to zero* and calculating *responsibility message r(i,k)*. The *responsibility message r(i,k)* represents how well suited data point k is as the exemplar of i.

$$r(i,k) \leftarrow s(i,k) - \max_{k's.t.k' \neq k} \{a(i,k') + s(i,k')\} \tag{4.5}$$

Availability message a(i,k) is updated based on the *responsibility messages* calculated prior in Equation 4.5. *Availability messages* are sent from candidate exemplar k to data point i suggesting whether k will be a good exemplar for i.

$$a(i,k) \leftarrow \min \left\{0, r(k,k) + \textstyle\sum_{i's.t.i' \notin \{i,k\}} \max\{0, r(i',k)\}\right\} \tag{4.6}$$

Self-availability is updated using:

$$a(k,k) \leftarrow \sum_{i's.t.i' \neq k} \max\{0, r(i',k)\} \tag{4.7}$$

The next iteration uses the new value of *availability messages* generated in Equation 4.6 to update *responsibility messages* in Equation 4.5. This procedure is continued iteratively: responsibilities are first updated given the availabilities (these are initially set to zero), availabilities are updated given the newly calculated responsibilities, and finally the exemplars for each iteration are calculated.

At any point in time, the exemplar for data point i can be found by combining the responsibilities and availabilities.

$$exemplar(i) \leftarrow \underset{k}{\operatorname{argmax}}\{a(i,k) + r(i,k)\} \tag{4.8}$$

Data points that have no similarity (a similarity of zero in our experiments) do not need to send messages between them. If data points do not share any similarity, these points should never be exemplars for each other.

Once the exemplars have converged, the affinity propagation algorithm terminates. As responsibilities and availabilities for each data point are updated, oscillations can occur. To prevent oscillations, we use a damping factor, λ, with a value between 0 and 1.

$$\mu_{new} = \lambda * \mu_{old} + (1 - \lambda) * (\mu_{new}) \tag{4.9}$$

whereby each message is set to λ times it previous value plus 1 minus λ times its new value. Here μ is *a(i,k)* or *r(i,k)*.

4.3 Architecture

In this section, we explain the architecture of our distributed implementation of SpamTracker [34]. This architecture can be used by existing domain mail servers to perform collaborative spam filtering. Figure 4.2 provides a high level overview of the architecture

along with the overlay network topology that is used for sharing information. The architecture diagram shows a logical separation between super-nodes that are connected in a peer-to-peer fashion to aggregate information for clustering. The super-nodes form the backend of the system. We can have settings where domain mail servers or nodes at an enterprise which have relationship with the domains, can act as super nodes. The super-nodes collectively aggregate the information about the traffic of spammers and perform clustering. The cluster information is sent back to the domain mail servers. Classification of new received emails is performed locally at the domain mail servers.

Currently we have implemented the backend of the system that has a set of super-nodes running the clustering algorithm in a distributed fashion. Section 4.3.1 explains the backend architecture and gives details on how clustering is performed in the system. In Section 4.3.2 we explain how the domain mail servers can use the cluster information to classify spammers. The distributed implementation of the backend is discussed in section 4.3.3.

4.3.1 Clustering

Our implementation uses affinity propagation [14] to calculate clusters that are refined iteratively by performing clustering over the information periodically collected from domain mail servers. This information for clustering includes details regarding the frequency of emails a domain rejects from spammers. This can be obtained from domain mail servers that use conventional filters to filter spam [41] . The list can also be enriched from user feedback on spam emails.

We use the dataset that *SpamTracker* [34] used for clustering, which contains the information regarding the frequency of emails from spammers that a domain receives. The dataset is from an email hosting provider's decisions about early mail rejects from hundreds of domains. This data contains the received time of a given email, anonymized sender of the email, accept or reject decision and the targeted domain, and lasts for a period of one month, from March 1^{st} 2007 to March 31^{st} 2007.

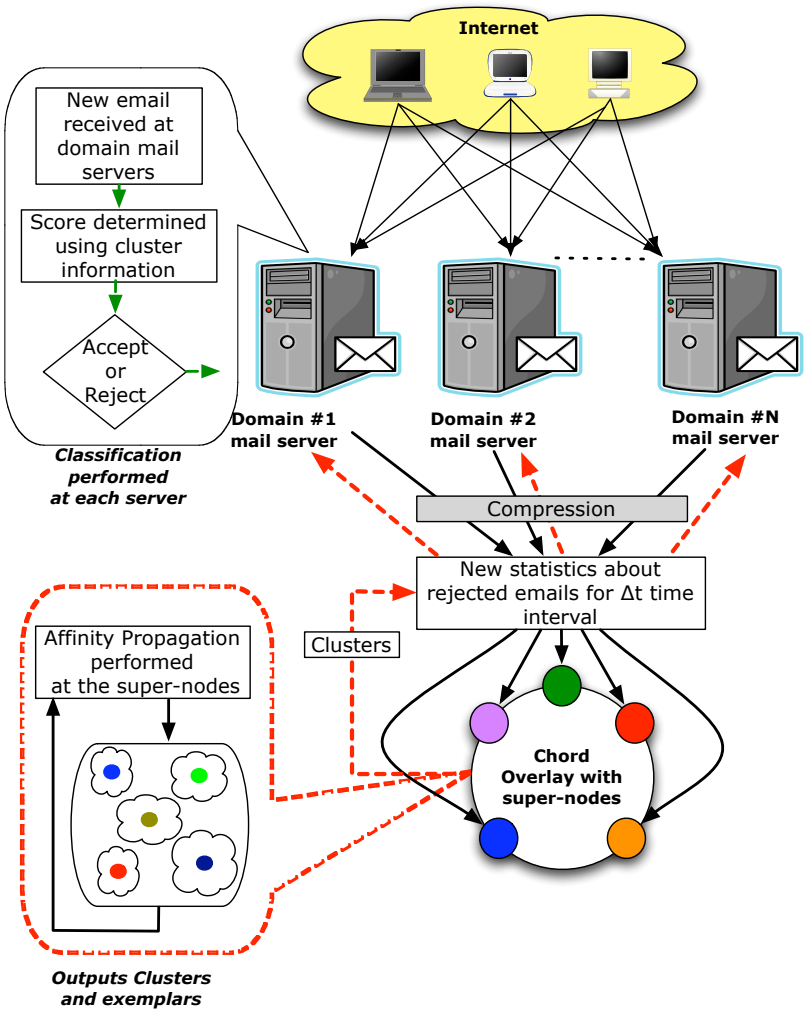


Figure 4.2. High level overview of the Distributed Collaborative Spam Detection System Architecture. The compression block is not implemented currently.

The spammer's activity at a particular domain is sent to a super-node that is geographically close to the domain mail server. We have chosen the Chord lookup protocol[45] to locate the super-node that corresponds to the location where the data of the sender IP address is aggregated. Chord maintains the nodes in a ring-based network and uses the ring geometry for efficient routing. The super-nodes in Chord aggregate the sender IP address activity across all domains. The aggregated information for each sender IP address represents the row in matrix N (Equation 4.1) for the IP address. Section 4.5.1 discusses

details on how efficient aggregation can be accomplished with domain mail servers acting as super-nodes connected in a peer-to-peer network.

After the aggregation has been performed, the similarity of a sender IP address's sending pattern with other IP address is calculated (Equation 4.2). This similarity information forms the basis for clustering. The clusters, calculated by running distributed affinity propagation, are sent back to the domain mail servers, communicating the sending patterns of spammers.

The clusters are refined periodically after a time interval $\triangle t$. We choose $\triangle t$ to be six hours (the same value as chosen in SpamTracker [34]) but this can be made configurable and changed randomly to overcome evasion [1]. Email rejection information for $t + \triangle t$ is collected and sent to the super-nodes for updating the clusters.

4.3.2 Classification

Whenever a domain mail server receives an email it creates or appends the sending information for the IP address to its own logs. The server calculates in real-time the score of the IP address' sending pattern with respect to the recent cluster information stored locally. The magnitude of the score S computed using Equation 4.4 determines how closely the sending pattern of the IP address matches a spammer's sending pattern. Each domain mail server can incorporate its own threshold value for the score and decide actions to be performed against emails that have a higher score than the threshold.

4.3.3 P2 Distributed Implementation

Our work involves implementation of the backend of the system described in Section 4.3. The backend has a set of super-nodes running the affinity propagation clustering algorithm in a distributed fashion. Affinity propagation is implemented in Overlog and is run on all super-nodes. Refer to Appendix C for the detailed Overlog.

Each super-node stores information regarding the similarity of a set of IP addresses.

[1] Refer to Section 4.5 for further discussion on evasion

The similarity is calculated using Equation 4.2. These super-nodes send messages, locally or to remote nodes that have IP addresses with similar sending behavior to the local IP addresses.

To reduce the amount of information aggregated at the super-nodes, we took a greedy approach for performing *cluster compression*. This approach reduces the number of IP addresses that have to be clustered and thus reduces the computation performed for clustering. Each row in the complete matrix N (Equation 4.1) is classified with previously computed cluster averages. Rows having high scores (Equation 4.4) are removed from the matrix and their sending information is incorporated into the previous clusters. This is computed by averaging the new rows and the previous clusters (Equation 4.3) to get the new updated clusters. A new matrix is formed, which includes the new clusters along with a mutually exclusive random sample of IP addresses that sent email during this same $\triangle t$ interval. This mxd matrix (subset of nxd matrix) is used for affinity propagation as before.

4.4 Evaluation

In this section, we evaluate our distributed declarative implementation of SpamTracker [34]. All experiments have been performed on the Emulab testbed [47].

Our main goal here is to examine the conciseness of our distributed clustering implementation as well as verify whether we get similar score distribution (for accepted and rejected IP addresses) as the SpamTracker implementation of Feamster et al. [34] after performing classification (Equation 4.4). This will provide an initial evidence that our implementation performs similar to the centralized implementation of Feamster et al.

Our second goal is to evaluate the scalability of our distributed implementation. In order to make the distributed implementation practical it is required to have low bandwidth consumption. Due to the limitations of the P2 prototype and issues in selecting the appropriate clustering algorithm for distributed settings, we are partially successful in

achieving our second goal. We discuss issues related to the algorithm selected in Section 4.5 and present how these may be addressed to make the application more scalable.

Due to the limitations mentioned above, we could not perform large experiments. In order to further verify that affinity propagation performs as well as spectral clustering we run the centralized implementation of affinity propagation (implemented in MATLAB)[2] on the SpamTracker data [34] from March 1^{st} 2007 to March 15^{st} 2007. We then do classification on the next 15 days data based on the clusters from affinity propagation to find the percentage of new detected spammers. This exercise helps us to check whether affinity propagation is able to detect spammers similar to the spectral clustering algorithm in SpamTracker [34].

For all experiments the dataset used to determine the sending pattern of different IP addresses is the same dataset as used by the SpamTracker application [34]. This data contains the received time and anonymized sender of a given email, its accept or reject decision and the targeted domain. This data spans a period of one month from March 1^{st} 2007 to March 31^{st} 2007. The sender IP addresses associated with a *rejected* email are considered spammers while *accepted* email sender IP addresses are considered benign. In this dataset we see some domains receive less emails in the entire month of March. We removed domains that receive fewer than fifty emails in the entire month leaving us with just over two hundred domains.

Section 4.4.1 explains the experiment evaluating the score distribution. In Section 4.4.2 we evaluate the affinity propagation algorithm to see whether it can perform as well as the spectral clustering algorithm in terms of detecting spammers.

4.4.1 Score Distribution Experiment

The first experiment evaluates our distributed implementation by running P2 on multiple Emulab [47] nodes. The Emulab nodes emulate super-nodes in our architecture (Figure 4.2). Currently, the distribution of IP addresses across super-nodes is provided

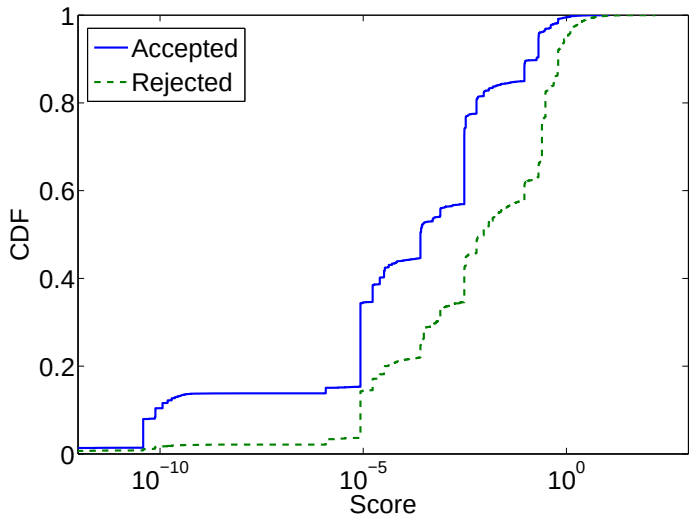
[2]This implementation is free to view and download at http://www.psi.toronto.edu/afinitypropagation

externally rather than assigning a key to each IP address and using Chord lookup protocol [45] to locate the super-node that corresponds to the location where the data of the sender IP address is aggregated. Similarly, the similarity calculations are calculated externally and provided to the super-nodes.

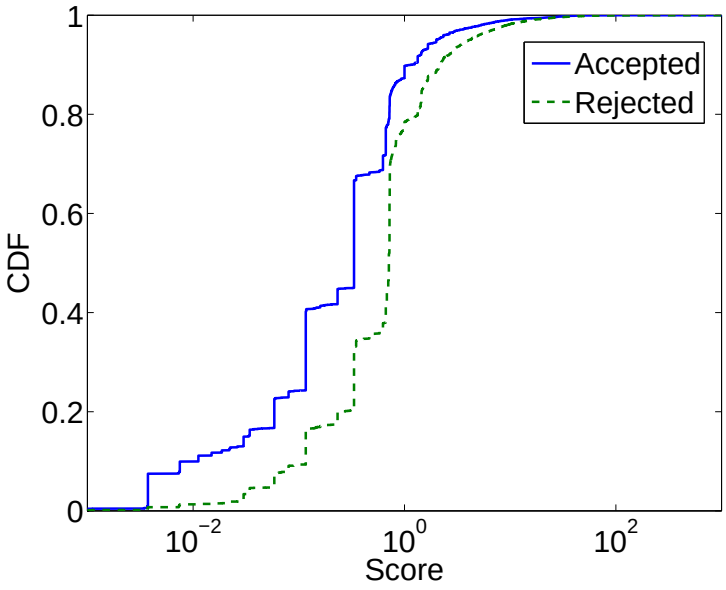
Due to a bug in P2 as well as the scalability issues discussed in Section 4.5, we were not able to perform experiments on more than 5 nodes that have a total of 30 IP addresses to cluster. The bug was due to non-optimized implementation of aggregation that became a bottleneck for our implementation. We discuss this issue in detail in Section 4.5.3 and propose a fix as well.

In order to overcome this issue, we use the random sampling and greedy approach of Section 4.3.3 for clustering a subset of the data in a six hour interval. Random sampling is used to get random samples of 30 IP addresses from a six hour dataset. The sampled IP addresses are clustered and the greedy approach is used to remove non-clustered IP addresses in the 6 hour dataset that are similar to the generated clusters (they give high scores when classified with the clusters). We incorporate these IP addresses in the cluster and update the cluster average. The cluster averages are used along with the next random sample to refine the clusters. These steps are performed iteratively to cluster approximately 3500 spammers out of $\sim 10K$ IP addresses in the six hour interval.

We picked up the complete N matrix from the next six hour interval and classified (4.4) it against the clusters calculated from the subset of the data in the previous six hour interval. Figure 4.3(a) shows the distribution of scores for both, the list of *accepted* and *rejected* IP addresses in the next six hour interval. We also performed the same experiment with the centralized implementation of affinity propagation and instead of clustering the same subset, we clustered all the IP addresses in the previous six hours. The score distribution of the centralized experiment is shown in figure 4.3(b).



(a) Distributed Experiment



(b) Centralized Experiment

Figure 4.3. The distribution of scores for distributed and centralized experiment. The distributed experiment uses clusters generated from a subset (3500 IP addresses out of ~10K) of the 6 hour data while the centralized experiment uses clusters generated from entire data (~10K IP addresses) in the six hour time. The blue solid line represents the *accepted* IP addresses and the green dashed line represents the *rejected* IP addresses.

We observe from the graph in Figure 4.3 that the magnitude of the scores in the distributed experiment (Figure 4.3(a)) is smaller in magnitude compared to the scores in the centralized experiment (Figure 4.3(b)). This is because the cluster information in the distributed experiment is not as rich as in the centralized experimental run (due to small number of IP addresses being clustered in the distributed experiments) and thus have small weight compared to the centralized experiment cluster (that include information of all the IP addresses).

In spite of different score magnitudes, the score distribution are similar in both the experiments and to the results in [34]. The similarity is in terms of the *Rejected* IP addresses having larger score magnitude compared to the *Accepted* IP addresses. This is because rejected IP addresses have similar sending pattern to the sending pattern of the spammer IP addresses that were clustered, while the accepted IP addresses sending pattern do not match with that of the clustered spammers.

4.4.2 Spam Detection Experiment

Since we use a different clustering algorithm than that used by Feamster et al. [34], we need to verify whether our distributed implementation can detect new spammers. To perform this evaluation, we run a complete experiment in a centralized manner to avoid the performance and scalability issue. In this experiment, we first run a single round of clustering for a six hour time interval. We combined the cluster averages, which were generated from the six hour interval, with the next six hours to generate a new set of clusters using affinity propagation. This is repeated until we covered a time period of 15 days, from March 1^{st} 2007 through March 15^{th} 2007.

We then classified the sending pattern of all the IP addresses from March 16^{th} 2007 through March 31^{st} 2007, using the generated clusters. Similar to the implementation of Feamster et al. [34], we classify all accepted IP addresses with a score of equal to or more than five as spammers. Figure 4.3(b) shows that a lot of IP addresses get a score above this threshold.

Feamster et al. use a blacklist to verify whether the accepted IP addresses that were classified as spammers (because they received score above five) were really spammers. Since we do not have access to an IP blacklist, we need a scheme to determine if an accepted IP address that gets a high score in our implementation, is actually a spammer. On examining the dataset, we saw that quite a few of the IP addresses that were accepted between March 1^{st} 2007 to March 15^{th} 2007, were rejected between March 16^{th} 2007 to March 31^{st} 2007. Intuitively, this can happen because the email from the spammer went undetected and was detected later because it got added to a blacklist. Morever, Feamter et al. have discussed in [34] that the data provider estimates that as much as 15% of accepted email is spam.

We use the technique discussed above to check whether we detected any new spammers. Our results indicate that almost 30% of the *accepted* IP addresses that get scores of magnitude greater than five become rejected later in the month. This shows that affinity propagation works well and is able to cluster spammer sending patterns that aids in detecting new spammers. We cannot compare our results with the centralized implementation of Feamster et al. because they use an IP blacklist to find the percentage of new spammers detected.

It is not possible to estimate false positives or negatives because of no ground truth. It can be the case that the rejected IP addresses may have been misclassified by the email provider of the data. Also, an IP address is not a permanent identifier because many machines obtain IP addresses from dynamic address pools. This can cause aliasing issue where a single machine may be associated with different IP addresses over time or a single IP address may represent multiple machines.

4.5 Issues and Future Work

In this section we present the issues related to our distributed implementation and how these issues may be resolved.

The current implementation and design of our system shows that a declarative framework can be used to implement algorithms that perform distributed spam filtering. As we see in the evaluation, the results of the distributed implementation of *SpamTracker* [34] using P2 show signs of detecting spam and give a similar score distribution to that of the Feamster et al. SpamTracker implementation. Even though the system proves to have the capability of being used for spam detection, there are issues that need to be resolved to make it scalable and practical. We discuss these issues in the following sections.

4.5.1 Clustering Sender Matrix

The *Spamtracker* system [34] clusters sender IP addresses to find spammer sending patterns. The sending pattern of a sender IP is spread across multiple domain mail servers. In terms of the reasoning graph, each random variable in the graphical model is the sending pattern associated with a sender IP and is spread across multiple nodes (domains). We need to aggregate the value of each random variable, which in turn leads to an extra communication overhead and makes the application less scalable. We discuss some techniques that can be used to address this issue in Section 5.2.

4.5.2 Algorithm Selection

The affinity propagation algorithm fits very well in our distributed inference model because it is a message passing algorithm and can be easily modeled on the reasoning graph. We demonstrate a concise declarative implementation of the algorithm using Overlog [3].

Even though we implemented the algorithm easily in Overlog and it gives similar results as the spectral clustering algorithm of SpamTracker [34], there lies an inherent drawback that makes it unsuitable for distributed systems with large data. As mentioned before in Section 4.2.3, the variables that are clustered using *affinity propagation* are sender IP addresses. The algorithm iteratively sends messages between each variable until the clusters are determined. Thus, in case of n variables (sender IP addresses) we require n^2

[3]Refer Appendix C for Overlog implementation of affinity propagation

communication and this becomes a major performance concern when *n is large*. The spam dataset we used had roughly *~10K* sender IP addresses in a *6 hour* time interval.

4.5.3 Aggregation Optimization

Another issue that our implementation faced was the non-optimized implementation of aggregation. Current implementation of aggregation in P2 scans all the tuples in a table to calculate the result. This computation is performed every time an aggregation query is fired, even when the table has not changed. One optimization that can be done is to store the aggregation results in a cache and mark them dirty if the table gets updated. In case of no changes the aggregation results can be picked up from the cache instead of performing aggregation again. This approach can be written as a set of rules in Overlog. Cache updates can be done eagerly or lazily. The optimization we discussed is a case of *materialized view maintenance* [4, 17] in database terms, but the presence of recursive rules makes the implementation a little complicated.

4.6 Related Work

Multiple applications can benefit from our distributed inference framework. In this thesis, we considered a spam detection application that uses behavioral blacklisting.

Content-based or IP-based blacklisting techniques are traditional approaches used for spam filtering. Content-based filtering techniques evaluate the content of the message to classify spam. The techniques tend to either use Bayesian filtering techniques to classify email as spam [15, 28, 36] or use a signature or checksum [6] of the message to compare it against a spam database on the Internet. Both these techniques require training data that captures the contents in the spam. Moreover, spammers dynamically add textual polymorphism to their spam to evade such filters.

Blacklists [43, 42, 3, 19] contain IP addresses that are considered to be associated with spammers. Many spam filters [41, 2] use these lists along with other content-based

schemes. These lists can be static or dynamic and are stored in databases that can be queried [19]. Reactive blacklists try to update the list based on tracking whether an IP address is associated with a spammer and update the list as the IP addresses are altered [43, 42, 3].

Pure content-based filtering techniques have become ineffective because spammers change email content frequently by using images in the email, or by sending well crafted emails that affect the Bayesian learner or classifier [48, 30]. IP address-based blacklists work for fixed IP addresses, but become outdated quickly, requiring frequent updates. These issues have led to detecting spam based on an IP address' sending or traffic pattern [34, 44]. Ramachandran et al. argues that spammers do not change their sending pattern frequently, thus making it efficient to detect them based on their sending behavior [34]. The sending behavior of a spammer is determined by the frequency of emails sent by the spammer to each domain. A similar concept has been used in SpamHINTS [44, 7, 8]. SpamHINTS uses heuristics related to *Simple Mail Transfer Protocol* (SMTP) sessions of a sender. These include measuring delivery failures and analyzing delivery failure messages.

Characterization studies [33] have observed that the local view of malicious spamming activity remains undetected due to the low volume of activity. This raises the need to aggregate spammers' activity across domains. A popular approach uses clustering to identify groups of spam messages or hosts. Li et al. cluster spammers based on the URL present in the emails and find huge clusters of spammers having the same URL [23]. The method proposed by Anderson et al. identifies clusters of web servers that host graphically similar websites linked from the messages [1]. The graphic similarity between websites is found using a technique called *image shingling*. This method is used to find web servers hosting phishing websites. Both of these methods use email contents for clustering. Clustering has also been used by Ramachandran et al. [34] to identify spammers based on their sending pattern.

Most of the approaches discussed above [34, 23, 1] are centralized and exhibit disadvantages such as scalability and single point of failure. The schemes established by Damiani et al. and Brodsky outline a system that collaboratively shares information to

detect spammers [11, 5] . Damiani et al. propose comparing incoming messages to known spam messages classified by an automatic mechanism or by final recipients [11]. Techniques like message digests, URLs in the email and originating mail servers can be used for comparison. Brodsky et al. present an approach that counts the quantity of emails to determine whether the emails were sent from spamming bots [5].

4.7 Summary

In this chapter, we took spam detection as an application of distributed inference. We picked the centralized and offline scheme of spam detection from the work of Feamster et al. [34] to see whether declarative programming makes it easy to build a distributed implementation of the same. In summary, we made the following observations from this exercise:

- We implemented a distributed version of the affinity propagation algorithm for clustering the sending pattern of spammers. Our declarative implementation of the algorithm is concise, with just 14 rules in Overlog[4].
- We were able show that our distributed implementation gave similar score distribution as the centralized implementation of Feamster et al. and was able to detect spammers, although our implementation is less scalable partly due to the non-optimized implementation of aggregation in the P2 prototype, and partly because affinity propagation does not scale in communication complexity.

Lessons learned from our first attempt towards using distributed inference for collaborative spam filtering include exploiting natural data partitioning to reduce aggregation, selecting inference algorithms that have less communication costs and sharing summaries instead of data.

[4]The Overlog code is listed in Appendix C

Chapter 5

Discussion

In this chapter we present the contributions made in this thesis and also discuss future work in terms of applying distributed inference for scalable collaborative spam detection.

5.1 Contribution

Our work on the declarative implementation of distributed inference has two high-level goals: First, through the use of a declarative language we aim to greatly simplify the process of specifying, implementing and customizing distributed inference algorithms. Second, we aim to provide ease of development of distributed inference techniques for network monitoring problems.

We summarize the contributions of our declarative framework for distributed inference as follows:

1. We define a layered graph architecture that uses a combination of **overlays** and **declarative programming** [25] to design distributed inference algorithms. This architecture is used to implement a set of existing inference algorithms like the **Junction Tree Inference**, **Loopy Belief Propagation** and **Affinity Propagation** in a distributed fashion. The declarative language **Overlog** aided in concise implementation of these algorithms often resulting in 4 times savings in code sizes. In addition to conciseness,

it also helped in customizing existing algorithms for example, we easily customized the loopy belief propagation algorithm to introduce randomized message scheduling scheme for faster convergence.

2. Our first step towards applying distributed inference on a real-world problem involved implementing a declarative version of the sensor calibration problem in sensor networks by Paskin et al. [31]. We easily implemented the two graphs: spanning tree and junction tree, using our proposed graph architecture (Figure 2.1) for distributed inference. Our declarative implementation has 4 times less code than the Lisp implementation of Paskin et al.

3. In order to evaluate the usefulness of distributed inference we applied it to a spam detection application. Overlog aided in a concise implementation of the affinity propagation clustering algorithm. This algorithm was used for clustering spammers with similar behavior. This was our first step towards evaluating the practicality of distributed inference. We were partly successful on this front since we could show that our implementation performed similar to the centralized implementation of Feamster et al. [34] and was able to detect spammers. Some major issues that need to be resolved include: selection of an inference algorithm that has less communication overhead and exploiting natural partitioning of data. We discuss future work in the direction of building a scalable spam detection system in the next section.

5.2 Future Directions

We believe that the distributed inference framework proposed in this thesis can have an impact in multiple applications, one of them being network monitoring. We applied our declarative distributed inference architecture to two network monitoring applications: sensor calibration in sensor networks [31] and behavioral spam detection [34]. This thesis is a first step towards building a system for distributed inference using declarative networking and applying it to network monitoring problems. In general we are optimistic that this framework can provide a basic tool that a diverse set of applications can take benefit from.

As we discussed in Section 4.5.1, our distributed implementation of the behavioral spam detection application [34] had random variables (sending pattern of a sender IP) spread across nodes (domain mail servers) in the network. As a result of this we needed to collectively aggregate the value of each random variable from multiple domains, which in turn lead to an extra communication overhead and made the application less scalable.

In the simplest case, distributed inference should be implemented in a context where each random variable is only observed at a single node. In this case we avoid the aggregation overhead. Our declarative implementation of the sensor networks application discussed in Section 3.1.1 had each random variable (sensor measurement) observed at a single node (sensor nodes) in the network. Thus, the application did not have the communication overhead of aggregating the observed random variables and was scalable enough to perform as well as the Lisp implementation of Paskin et al.

Here we discuss few approaches that can be investigated in the future to reduce or avoid the communication overhead associated with aggregating random variables:

1. **Minimizing communication:** *Temporal compression* [40, 18] heuristics can be used to reduce the amount of information that has to be shared for computing each random variable after a time interval $\triangle t$. Temporal compression involves sharing information related to a variable only if its value at each domain changes beyond a certain threshold.

2. **Active learning:** Active learning is a machine learning technique that reduces the number of random variables (that are labelled in our case as spam or non-spam) required for training a good classifier. This technique has a natural application in spam detection since it picks important labeled information to build an effective classifier. Active machine learning has been used in the past for spam filtering [39]. In our distributed SpamTracker implementation, active learning may help in reducing the communication costs incurred in computing the random variables by only aggregating the more important ones.

3. **Hierarchical clustering:** Instead of computing the random variables and then per-

forming clustering, we can take a hierarchical approach. As mentioned in Section 4.2.1, our data is a matrix N nxd, where n are the random variables (sender IPs) and d are the nodes (domain mail servers) in the network. Each domain mail server d observes a column of the matrix N, which means that the data is partitioned vertically. Hierarchical clustering can be used to generate clusters from these vertical partitions (a column of the matrix N) residing locally at each domain. These local clusters from each domain can be combined to get global clusters. Johnson et al. [21] have proposed hierarchical clustering for clustering vertically partitioned data.

4. **Different graphical model:** We can use a different graphical model where each random variable is only observed at a single node. In this case we incur communication costs for performing inference and avoid the aggregation overhead. In the context of SpamTracker [34], we need to identify locally observed features (random variables) associated with emails received at each domain. Prof. Feamster and his students have been investigating ideas to look at the local information associated with an email a domain receives from a sender. This information can be: geodesic distance between the sender and receiver, AS number of sender, status of the sender's email service ports and the received time of the email. Distributed inference techniques can be used for designing algorithms that can run locally at each domain and can calculate an approximate summary of the locally observed features, which can then be combined to reach global conclusions.

5.3 Closing

The distributed nature of automatically-generated information is present both in the physical world and in computer networks. We believe that inference techniques can be used to deal with locally observed information and can be used to assemble them to reach global conclusions. This thesis uses a declarative language that can greatly simplify the process of specifying, implementing and deploying distributed inference algorithms. We demonstrate that the Overlog implementations provide a natural and compact way of

expressing a variety of well known inference algorithms. We take a first step towards using distributed inference techniques for spam detection. Even though our results are mixed, we believe that declarative scalable spam detection can be designed with less communication intensive inference algorithms and exploitation of the natural partitioning of data.

Bibliography

[1] David S. Anderson, Chris Fleizach, Stefan Savage, and Geoffrey M. Voelker. Spamscatter: Characterizing Internet Scam Hosting Infrastructure. In *USENIX Security*, 2007.

[2] Mail Avenger. `http://www.mailavenger.org/`.

[3] Realtime URI Blacklist. `http://www.uribl.com/`.

[4] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently Updating Materialized Views. *SIGMOD Rec.*, 15(2):61–71, 1986.

[5] Alex Brodsky and Dmitry Brodsky. A Distributed Content Independent Method for Spam Detection. In *HotBots'07: Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, pages 3–3. USENIX Association, 2007.

[6] Distributed Checksum Clearinghouse. `http://www.rhyolite.com/anti-spam/dcc/`.

[7] Richard Clayton. Stopping Spam by Extrusion Detection. In *CEAS*, 2004.

[8] Richard Clayton. Stopping Outgoing Spam by Examining Incoming Server Logs. In *CEAS*, 2005.

[9] R. G. Cowell, A. P. Dawid, S. L. Lauritzen, and D. J. Spiegelhalter. *Probabilistic Networks and Expert Systems*. Springer-Verlag, Berlin-Heidelberg-New York, 1999.

[10] Christopher Crick and Avi Pfeffer. Loopy Belief Propagation as a Basis for Communication in Sensor Networks. In *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence (UAI-2003)*, San Francisco, 2003. Morgan Kaufmann Publishers, Inc.

[11] Ernesto Damiani, Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. P2P-Based Collaborative Spam Detection and Filtering. In *P2P '04: Proceedings of the Fourth International Conference on Peer-to-Peer Computing*, pages 176–183, Washington, DC, USA, 2004. IEEE Computer Society.

[12] Amol Deshpande, Carlos Guestrin, Samuel Madden, Joseph M. Hellerstein, and Wei Hong. Model-driven Data Acquisition in Sensor Networks. In *Proceedings International Conference on Very Large Data Bases (VLDB)*, 2004.

[13] G. Elidan, I. Mcgraw, and D. Koller. Residual Belief Propagation: Informed Scheduling for Asynchronous Message Passing. In *Proceedings of the Twenty-second Conference on Uncertainty in AI (UAI)*, Boston, Massachussetts, July 2006.

[14] Brendan J. J. Frey and Delbert Dueck. Clustering by Passing Messages Between Data Points. *Science*, 315:972–976, February 2007.

[15] P. Graham. Better Bayesian Filtering `http://www.paulgraham.com/better.html`, 2003.

[16] Sergio Greco. Dynamic Programming in Datalog with Aggregates. *IEEE Trans. Knowl. Data Eng.*, 11(2):265–283, 1999.

[17] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 157–166, New York, NY, USA, 1993. ACM.

[18] L. Huang, X. L. Nguyen, M. Garofalakis, J. Hellerstein, M. Jordan, A. D. Joseph, and N. Taft. Communication-Efficient Online Detection of Network-Wide Anomalies. In *Proceedings of the 26th IEEE Conference on Computer Communications*, 2007.

[19] A Iverson. DNSBL Resource Blacklist Statistics Center `http://stats.dnsbl.com/`.

[20] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and Jr. James W. O'Toole. Overcast: Reliable Multicasting with an Overlay Network. In *Symposium on Operating System Design and Implementation (OSDI)*, 2000.

[21] Erik L. Johnson and Hillol Kargupta. Collective, Hierarchical Clustering from Distributed, Heterogeneous Data. In *Revised Papers from Large-Scale Parallel Data Mining, Workshop on Large-Scale Parallel KDD Systems, SIGKDD*, pages 221–244, London, UK, 2000. Springer-Verlag.

[22] Michael Kearns, Jinsong Tan, and Jennifer Wortman. Privacy-Preserving Belief Propagation and Sampling. In J. C. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems 20*. MIT Press, Cambridge, MA, 2008.

[23] F. Li and M.-H. Hseih. An Empirical Study of Clustering Behavior of Spammers and Group-Based Anti-Spam Strategies. In *CEAS*, 2006.

[24] Boon T. Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing Declarative Overlays. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 75–90, New York, NY, USA, 2005. ACM Press.

[25] Boon T. Loo and Joseph Hellerstein. Declarative Routing: Extensible Routing with Declarative Queries. *SIGCOMM 2005*, Sep 2007.

[26] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative Networking: Language, Execution and Optimization. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 97–108, New York, NY, USA, 2006. ACM.

[27] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, 2002.

[28] T.A Meyer and B Whateley. SpamBayes: Effective Open-Source, Bayesian Based, Email Classification. In *CEAS*, 2004.

[29] Kevin P. Murphy, Yair Weiss, and Michael I. Jordan. Loopy Belief Propagation for Approximate Inference: An Empirical Study. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, pages 467–475, 1999.

[30] Blaine Nelson, Marco Barreno, Fuching Jack Chi, Anthony D. Joseph, Benjamin I. P. Rubinstein, Udam Saini, Charles Sutton, J. D. Tygar, and Kai Xia. Exploiting Machine Learning to Subvert Your Spam Filter. In *Proceedings of the First Workshop on Large-scale Exploits and Emerging Threats (LEET)*, 2008.

[31] Mark Paskin, Carlos Guestrin, and Jim Mcfadden. A Robust Architecture for Distributed Inference in Sensor Networks. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, page 8, Piscataway, NJ, USA, 2005. IEEE Press.

[32] K. H. Plarre and P. R. Kumar. Extended Message Passing Algorithm for Inference in Loopy Gaussian Graphical Models. *Ad Hoc Networks*, 2(2):153–169, April 2004.

[33] Anirudh Ramachandran and Nick Feamster. Understanding the Network-level Behavior of Spammers. In *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 291–302, New York, NY, USA, 2006. ACM.

[34] Anirudh Ramachandran, Nick Feamster, and Santosh Vempala. Filtering Spam with Behavioral Blacklisting. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 342–351, New York, NY, USA, 2007. ACM.

[35] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A Scalable Content-Addressable Network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, volume 31, pages 161–172. ACM Press, October 2001.

[36] Mehran Sahami, Susan Dumais, David Heckerman, and Eric Horvitz. A Bayesian Approach to Filtering Junk E-Mail. In *Learning for Text Categorization: Papers from the 1998 Workshop*, Madison, Wisconsin, 1998. AAAI Technical Report WS-98-05.

[37] J. Schiff, D. Antonelli, A. G. Dimakis, D. Chu, and M. J. Wainwright. Robust Message-Passing for Statistical Inference in Sensor Networks. In *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*, pages 109–118, 2007.

[38] Roman Schmidt and Karl Aberer. Efficient Peer-to-Peer Belief Propagation. In *OTM Conferences (1)*, pages 516–532, 2006.

[39] D. Sculley. Online Active Learning Methods for Fast Label-Efficient Spam Filtering. In *CEAS 2007: Fourth Conference on Email and Anti-Spam*, August 2007.

[40] Adam Silberstein, Gavino Puggioni, Alan Gelfand, Kamesh Munagala, and Jun Yang. Suppression and Failures in Sensor Networks: A Bayesian Approach. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 842–853. VLDB Endowment, 2007.

[41] SpamAssassin. http://spamassassin.apache.org/.

[42] Spamcop. http://www.spamcop.net/.

[43] Spamhaus. http://www.spamhaus.org/.

[44] SpamHINTS. http://www.spamhints.org/.

[45] Ion Stoica, Robert Morris, David Karger, Frans F. Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, October 2001.

[46] E. B. Sudderth, A. T. Ihler, W. T. Freeman, and A. S. Willsky. Nonparametric Belief Propagation. In *Computer Vision and Pattern Recognition, 2003. Proceedings. 2003 IEEE Computer Society Conference on*, volume 1, pages I–605–I–612 vol.1, 2003.

[47] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *In Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270. USENIX Association, 2002.

[48] Gregory L. Wittel and S. Felix Wu. On Attacking Statistical Spam Filters. In *Proceedings of the First Conference on Email and Anti-Spam (CEAS)*, 2004.

Appendix A

Junction Tree Inference

Here we provide the full Overlog specification for junction tree inference. The rules for the running intersection property are described in Figure A.1 while the inference rules are present in Figure A.2. The Overlog for spanning tree (used by junction tree) is in Figure A.3, A.4. For a detailed explanation of the distributed spanning tree algorithm please refer [31]. All the base relations in the Overlogs have been highlighted in italic font. The three layers: spanning tree, running intersection property (junction tree) and inference work together and get updated in case of node/link failures and additions.

As per Equation 3.2, Junction Tree Inference message computation includes two factors: local potential and incoming message factors. The base relations that represent these factors are: *localFactor* represents local potential ψ_{C_i} and *incoming* represents the received messages μ_{ki}.

Functions *f_product* and *f_marginal* encapsulate the operations to be performed for generating messages μ_{ij}.

```
jtUpdate(@Node, Time) :-
        periodic(@Node, E, JT_EPOCH),
        Time := f_timerElapsed().

jtNbrUpdate(@Node, Nbr) :-
        jtUpdate(@Node),
        edge(@Node, Nbr).

/* Incoming message.  The message is deleted when the edge gets deleted.  */
R1: reachable(@Node, Nbr, Vars) :-
        localVars(@Node, Vars),
        edge(@Node, Nbr).

/* Union of incoming reachvar message.  Localvars are included*/
R2: reachvars(@Target, Node, a_UNION<Vars>) :-
        jtNbrUpdate(@Node, Target),
        reachable(@Node, Nbr, Vars),
        Nbr != Target.

/* Compute the clique at each node.  */
C1: cliqueRIP(@Node, a_UNION<Vars>) :-
        jtUpdate(@Node),
        reachvars(@Node, Nbr1, Vars1),
        reachvars(@Node, Nbr2, Vars2),
        Nbr1 != Nbr2,
        Vars := Vars1 & Vars2.

C2: clique(@Node, Vars) :-
        cliqueRIP(@Node, RipVars),
        localVars(@Node, LocalVars),
        Vars := RipVars | LocalVars.

/* Compute the separator.  */
separator(@Node, Nbr, Vars) :-
        reachable(@Node, Nbr, NbrVars),
        clique(@Node, MyVars),
        Node != Nbr,
        Vars := MyVars & NbrVars.
```

Figure A.1. Overlog for Running Intersection Property.

```
/***********************************
One round of updates
***********************************/
m1:  jtinfUpdate(@Node, Time) :-
         periodic(@Node, E, JTINF_EPOCH),
         Time := f_timerElapsed(),
         incoming(@Node, Node, _).  /* Each node must have a local factor.  */

m2:  nbrUpdate(@Node, Nbr) :-
         jtinfUpdate(@Node, _),
         edge(@Node, Nbr).

/* Update the separator set.  */
m3:  separatorSet(@Node, Nbr, a_mkList<Var>) :-
         nbrUpdate(@Node, Nbr),
         separator(@Node, Nbr, Var).

m4:  incoming(@Node, Node, Factor) :-
         localFactor(@Node, Factor).

/* Calculate the factors whose product forms the messge.  */
m5:  msgFactors(@Node, TargetNbr, a_mkList<F>) :-
         nbrUpdate(@Node, TargetNbr),
         incoming(@Node, Nbr, F), /* includes the local factor */
         Nbr != TargetNbr.

/* Compute the message.  */
m6:  message(@Nbr, Node, NewF) :-
         msgFactors(@Node, Nbr, MessageFactors),
         separatorSet(@Node, Nbr, Retain),
         FProd := f_product(MessageFactors),
         NewF := f_marginal(FProd, Retain).

/* The incoming message.  Message is deleted on edge deletion*/
m7:  incoming(@Node, Nbr, Factor) :-
         message(@Node, Nbr, Factor),
         edge(@Node, Nbr).

/*********************************
Calculate beliefs
*********************************/
m8:  beliefFactors(@Node, a_mkList<Factor>) :-
         jtinfUpdate(@Node, Time),
         incoming(@Node, _, Factor).

m9:  belief(@Node, Factor) :-
         beliefFactors(@Node, BeliefFactors),
         Factor := f_product(BeliefFactors).

m10:  factorCount(@Node, Count) :-
         beliefFactors(@Node, Factors),
         Count := f_size(Factors).
```

Figure A.2. Overlog for Junction Tree Inference.

```
/* Update each node's own root pulse time.  */
p1:  pulse(@Node, MYID, Time) :-
        periodic(@Node, E, ROUTING_EPOCH),
        Time := f_timerElapsed().

/* Insert the default pulse for a root we have never heard.  */
p2:  pulse(@Node, RootId, Pulse) :-
        config(@Node, _, _, RootId, Pulse),
        notin pulse(@Node, RootId,_).

/* Update the pulse for the current parent.  */
p3:  pulse(@Node, RootId, Pulse) :-
        config(@Node, Nbr, _, RootId, Pulse),
        parent(@Node, Nbr).

/* Update the root for the current parent.  */
root(@Node, RootId) :-
        config(@Node, Nbr, _, RootId, Pulse),
        parent(@Node, Nbr).

/*****************************************************************************
Update the internal state and configurations by selecting the new best parent.
*****************************************************************************/
/* Run a single update.  */
r1:  updateparent(@Node) :-
        periodic(@Node, E, ROUTING_EPOCH).

/* The cost of changing a parent.  */
r2:  newparent(@Node, a_max<InvCostParent>) :-
        updateparent(@Node),
        config(@Node, Nbr, NbrParent, NbrRootId, NbrPulse),
        pulse(@Node, NbrRootId, OldPulse),
        parent(@Node, OldParent),
        root(@Node, OldRootId),
        NbrRootId <= OldRootId, /* Neighbor has a better root */
        Nbr != OldParent, /* Neighbor is not already my parent */
        NbrPulse > OldPulse, /* Neighbor is not my descendant.  */
        NbrParent != Node, /* Neighbor is not my child:  avoid cycles */
        link(@Node, Nbr, _, PReceive),
        Cost := 1.0 / PReceive + ROUTING_SWITCH_COST,
        InvCostParent := f_cons(1.0 - Cost, Nbr).

/* The cost of keeping the parent */
r3:  newOldparent(@Node, NewParent, a_max<InvCostParent>) :-
        newparent(@Node, NewParent),
        config(@Node, Nbr, NbrParent, NbrRootId, NbrPulse),
        parent(@Node, Nbr),
        NbrRootId < MYID,
        NbrParent != Node,
        link(@Node, Nbr, _, PReceive),
        Cost := 1.0 / PReceive,
        InvCostParent := f_cons(1.0 - Cost, Nbr).
```

Figure A.3. Spanning Tree Overlog (Part I).

```
/* Local node is the root since no parent has a lower root.  */
r4:  bestparent(@Node, Node) :-
         newOldparent(@Node, NewParent, OldParent),
         f_size(NewParent) == 0,
         f_size(OldParent) == 0.

/* Select the best parent and update parent, root and pulse.  */
r5:  bestparent(@Node, Parent) :-
         newOldparent(@Node, NewParent, OldParent),
         CostParent := f_max(NewParent, OldParent),
         f_size(CostParent) > 0,
         Parent := f_last(CostParent).

bestParentInfo(@Node, Parent, RootId, Pulse) :-
         bestparent(@Node, Parent),
         config(@Node, Parent, _, RootId, Pulse).

bestParentInfo(@Node, Node, MYID, Pulse) :-
         bestparent(@Node, Parent),
         Parent == Node,
         pulse(@Node, MYID, Pulse).

parent(@Node, Parent) :-
         bestParentInfo(@Node, Parent, _, _).

root(@Node, RootId) :-
         bestParentInfo(@Node, _, RootId, _).

p4:  pulse(@Node, RootId, Pulse) :-
         bestParentInfo(@Node, _, RootId, Pulse).

/* Send a configuration message describing Node's state to all neighbors.  */
c1:  config(@Nbr, Node, Parent, RootId, Pulse) :-
         bestParentInfo(@Node, Parent, RootId, Pulse),
         linkEnabled(@Node, Nbr).

c2:  configBroadcast(@Base, Node, Parent, RootId, Pulse) :-
         bestParentInfo(@Node, Parent, RootId, Pulse),
         Base := BASE_ADDR.

/* Establish bidirectional edges, used by upper levels.  */
edge(@Node, Parent) :-
         parent(@Node, Parent).

edge(@Parent, Node) :-
         parent(@Node, Parent).

config_inserted(@Node, Nbr) :-
         config(@Node, Nbr, Parent, RootId, Pulse).

/* Update the link age.  */
link(@Node, Nbr, PSend, PReceive, Time) :-
         config_inserted(@Node, Nbr),
         link(@Node, Nbr, PSend, PReceive, _),
         Time := f_timerElapsed().
```

Figure A.4. Spanning Tree Overlog (Part II).

Appendix B

Loopy Belief Propagation

Here we list the Overlog for naive loopy belief propagation in Figure B.1,B.2 and randomized loopy belief propagation in Figure B.3,B.4,B.5. As per Equation 3.5, message computation includes three factors: local potential, edge potential and incoming message factors. The base relations that represent these factors are: *nodePotential* represents local potential ψ_s, *edgePotential* represents potential of the edge $\psi_{s,t}$ and *incoming* represents the received messages $\mu_{r,s}(y_s)$.

Functions *f_combineAll*, *f_collapse* and *f_normalize* encapsulate operations to be performed for generating messages $\mu_{s,t}(y_t)$.

```
/* Edges of the reasoning graph connected to a variable at this node */
rgEdge(@Node, Source, Target) :-
        localVariable(@Node, Source, _),
        rgEdgeInput(@Node, Source, Target).

/*********************************************************************
Compute the new messages
*********************************************************************/
/* Messages coming into node from nodes OTHER than Nbr */
bpUpdate(@Node, Source, Target) :-
        periodic(@Node, E, BP_EPOCH),
        localVariable(@Node, Source, _, _),
        rgEdge(@Node, Source, Target).

/* Factors whose product forms the messge.  */
messageFactors(@Node, Source, Target, a_MKLIST<Factor>) :-
        bpUpdate(@Node, Source, Target),
        incoming(@Node, OtherVar, Source, Factor),
        OtherVar != Target.

/* The result of a single unweighted update */
evalMessage(@Node, Source, Target, NewFactor) :-
        messageFactors(@Node, Source, Target, InFactors),
        nodePotential(@Node, Source, NodeFactor),
        edgePotential(@Node, Source, Target, EdgeFactor),
        F := f_combineAll(EdgeFactor, NodeFactor, InFactors),
        NewFactor := f_normalize(f_collapse(F, Target)).
```

Figure B.1. Overlog for Naive Loopy Belief Propagation (Part I).

```
/*****************************************************************
Send messages to neighbors
*****************************************************************/
l1:  messageEvent(@Node, TargetNode, Source, Target, NewFactor, Residual) :-
         evalMessage(@Node, Source, Target, NewFactor),
         variable(@Node, Target, TargetNode, _, _),
         message(@Node, Source, Target, OldFactor),
         Residual := f_norminf(NewFactor, OldFactor).

l2:  messageEvent(@Node, TargetNode, Source, Target, NewFactor, Residual) :-
         evalMessage(@Node, Source, Target, NewFactor),
         variable(@Node, Target, TargetNode, _, _),
         notin message(@Node, Source, Target, _),
         Residual := DEFAULT_RESIDUAL.

/* The latest residual for each message.  */
residual(@Node, Source, Target, Residual) :-
         messageEvent(@Node, _, Source, Target, _, Residual).

message(@Node, Source, Target, NewFactor) :-
         messageEvent(@Node, _, Source, Target, NewFactor, _).

incoming(@TargetNode, Source, Target, NewFactor) :-
         messageEvent(@Node, TargetNode, Source, Target, NewFactor, _).

/*****************************************************************
Calculate the belief
*****************************************************************/
/* Periodically trigger the belief update event.*/
beliefUpdate(@Node, Var) :-
         periodic(@Node, E, BP_EPOCH),
         localVariable(@Node, Var, _, _).

/* The list of all incoming factors.  */
beliefIncoming(@Node, Var, a_MKLIST<Factor>) :-
         beliefUpdate(@Node, Var),
         incoming(@Node, _, Var, Factor).

belief(@Node, Var, F) :-
         beliefIncoming(@Node, Var, IncomingFactors),
         nodePotential(@Node, Var, NodeFactor),
         F := f_normalize(f_combineAll(NodeFactor, IncomingFactors)).

beliefValues(@Node, Var, Values, Time) :-
         belief(@Node, Var, Factor),
         Values := f_values(Factor),
         Time := f_timerElapsed().
```

Figure B.2. Overlog for Naive Loopy Belief Propagation (Part II).

```
/* The edges of reasoning graph connected to a variable at this node:  Node, From, To */
mrfEdge(@Node, Source, Target) :-
        localVariable(@Node, Source, _),
        mrfEdgeInput(@Node, Source, Target).

/************************************************************************
Compute the new messages
************************************************************************/

/* Messages coming into node from nodes OTHER than Nbr */
bpUpdate(@Node, Source, Target) :-
        periodic(@Node, E, BP_EPOCH),
        localVariable(@Node, Source, _, _),
        mrfEdge(@Node, Source, Target).

messageFactors(@Node, Source, Target, a_MKLIST<Factor>) :-
        bpUpdate(@Node, Source, Target),
        incoming(@Node, IncomingVar, Source, Factor),
        IncomingVar != Target.

evalMessage(@Node, Source, Target, NewFactor) :-
        messageFactors(@Node, Source, Target, IncomingFactors),
        nodePotential(@Node, Source, NodeFactor),
        edgePotential(@Node, Source, Target, EdgeFactor),
        FactorList := f_cons(EdgeFactor, f_cons(NodeFactor, IncomingFactors)),
        IntermediateFactor := f_combineAll(FactorList),
        TargetVars := f_cons(Target, f_initlist()),
        NewFactor := f_normalize(f_collapse(IntermediateFactor, TargetVars)).

/************************************************************************
Perform a weighted update and compute the residuals
************************************************************************/
/* Compute the weighted sum of the new and previous message */
l1:  messageEvent(@Node, TargetNode, Source, Target, MixedFactor, Residual) :-
        evalMessage(@Node, Source, Target, NewFactor),
        message(@Node, Source, Target, OldFactor),
        variable(@Node, Target, TargetNode, _, _),
        normalizer(@Node, Normalizer),
        Residual := f_norminf(NewFactor, OldFactor),
        f_coinFlip(f_pow(Residual, BP_EXPONENT) / Normalizer) == 1,
        MixedFactor := f_weightedUpdate(OldFactor, NewFactor, BP_UPDATE_RATE).

l2:  messageEvent(@Node, TargetNode, Source, Target, NewFactor, Residual) :-
        evalMessage(@Node, Source, Target, NewFactor),
        notin message(@Node, Source, Target, _),
        variable(@Node, Target, TargetNode, _, _),
        Residual := DEFAULT_RESIDUAL.

/* The latest residual (whether a message was sent or not) */
residual(@Node, Source, Target, Residual) :-
        messageEvent(@Node, TargetNode, Source, Target, MixedFactor, Residual).
```

Figure B.3. Overlog for Randomized Loopy Belief Propagation (Part I).

```
/* A message sent to a neighbor.  */
message(@Node, Source, Target, NewFactor) :-
        messageEvent(@Node, _, Source, Target, NewFactor, _).

incoming(@TargetNode, Source, Target, NewFactor) :-
        messageEvent(@Node, TargetNode, Source, Target, NewFactor, _).

/*************************************************************************
Calculate the beliefs
*************************************************************************/
/* Periodically trigger the belief update event.
We cannot simply listen on incoming insertions here, since that means
that we may be double-counting the old and the new incoming messages.
*/
beliefUpdate(@Node, Var) :-
        started(@Node),
        periodic(@Node, E, BP_EPOCH),
        localVariable(@Node, Var, _, _).

/* The list of all incoming factors.  */
beliefIncoming(@Node, Var, a_MKLIST<Factor>) :-
        beliefUpdate(@Node, Var),
        incoming(@Node, _, Var, Factor).

belief(@Node, Var, F) :-
        beliefIncoming(@Node, Var, IncomingFactors),
        nodePotential(@Node, Var, NodeFactor),
        F := f_normalize(f_combineAll(f_cons(NodeFactor, IncomingFactors))).

beliefValues(@Node, Var, Values, Time) :-
        belief(@Node, Var, Factor),
        Values := f_values(Factor),
        Time := f_timerElapsed().
```

Figure B.4. Overlog for Randomized Loopy Belief Propagation (Part II).

```
/*************************************************************************
Calculate average residual value
*************************************************************************/
/* Compute the sum and count of residuals of messages sent to remote nodes.  */
localResidualCount(@Node, a_COUNT<Residual>) :-
        periodic(@Node, E, AGG_EPOCH),
        residual(@Node, _, Target, Residual).

localResidual(@Node, a_SUM<Term>, Count) :-
        localResidualCount(@Node, Count),
        residual(@Node, _, Target, Residual),
        Term := f_pow(Residual, BP_EXPONENT).

/* Update the local aggregate.  */
receivedResidual(@Node, Node, Sum, Count) :-
        localResidual(@Node, Sum, Count).

incomingResidual(@Node, Node, Sum, Count) :-
        receivedResidual(@Node, From, Sum, Count),
        From == Node.

/* Send an update to each neighbor in the spanning tree */ nbrUpdate(@Node, Nbr) :-
        localResidual(@Node),
        edge(@Node, Nbr).  /* the edge in the spanning tree */

receivedResidual1(@Node, Nbr, a_SUM<Sum>) :-
        nbrUpdate(@Node, Nbr),
        incomingResidual(@Node, Other, Sum, _),
        Other != Nbr.

receivedResidual(@Nbr, Node, Sum, a_SUM<Count>) :-
        receivedResidual1(@Node, Nbr, Sum),
        incomingResidual(@Node, Other, _, Count),
        Other != Nbr.

incomingResidual(@Node, Nbr, Sum, Count) :-
        receivedResidual(@Node, Nbr, Sum, Count),
        edge(@Node, Nbr).
estimatedResidual1(@Node, a_SUM<Sum>) :-
        localResidual(@Node),
        incomingResidual(@Node, Nbr, Sum, _).

estimatedResidual(@Node, Sum, a_SUM<Count>) :-
        estimatedResidual1(@Node, Sum),
        incomingResidual(@Node, Nbr, _, Count).

N: normalizer(@Node, Value, Time) :-
        estimatedResidual(@Node, Sum, Count),
        Count > 0,
        Value := Sum / Count,
        Time := f_timerElapsed().
```

Figure B.5. Overlog for Randomized Loopy Belief Propagation (Part III).

Appendix C

Affinity Propagation

Here we list the full Overlog specification for Affinity Propagation. All the base relations in the Overlog have been highlighted with a italic font. The Overlog has three main relations: *similarity*, *responsibility* and *availability*. The *similarity* relation stores the similarity each local IP address has with other IP addresses. *Responsibilities* and *availabilities* are initialized to zero and updated at regular time intervals equal to AP_EPOCH. Relation *sentResponsibility* is used to store the responsibilities sent by a node. This information saves us from doing a round-trip while calculating the exemplars.

For brevity, we have not shown the damping factor calculations, initialization messages and Chord integration for lookup of data location. Materialized table *variable* stores the location of the IP address. We have modified the algorithm to work for multiple variables per node and the *localVariable* table stores the information of IP addresses associated with a super-node.

Some abbreviations used in the Overlog are:

- A: Availability
- R: Responsibility
- S: Similarity
- CE: Candidate Exemplar
- AS: Availability + Similarity
- a_Max2Details<AS>: Aggregate function that returns the ordered pair of (CE, AS) with the largest two AS values

```
/*Responsibility updates for each local variable at a periodic time interval*/
rUpdate(@N, LocalVar) :-
        periodic(@N,E, AP_EPOCH),
        localVariable(@N, LocalVar, _).

/*Ordered set of similarity + availability (AS).*/
asSet(@N, LocalVar, a_Max2Details<AS>) :-
        rUpdate(@N, LocalVar),
        availability(@N, LocalVar, CE, A,),
        similarity(@N, LocalVar, CE, S),
        AS := A + S.

/*Responsibility calculations for a given r(i,k), i can be considered the LocalVar and
k is the variable CE. The CE here only relates to the candidate exemplars that do not
produce the maximum AS value for the LocalVar, i.  This is detailed in Equation 4.5*/

responsibilityEvent(@N, LocalVar, CE, NR) :-
        asSet(@N, LocalVar, Max2Details),
        similarity(@N, LocalVar, CE, S),
        MaxAS := f_removeLast(Max2Details),
        MaxCE := f_removeLast(Max2Details),
        NR := Similarity - MaxAS,
        CE != MaxCE.

/*Responsibility calcuations for a given r(i,k), i can be considered the LocalVar and k is
the variable CE. The CE here only relates to the candidate exemplars that does produce the
maximum AS value for the LocalVar, i.  This is detailed in Equation 4.5*/
responsibilityEvent(@N, LocalVar, CE, NR) :-
        asSet(@N, LocalVar, Max2Details),
        similarity(@N, LocalVar, CE, S),
        MaxAS := f_removeLast(Max2Details),
        MaxCE := f_removeLast(Max2Details),
        MaxAS2 := f_removeLast(Max2Details),
        MaxCE2 := f_removeLast(Max2Details),
        NR := S - MaxAS2,
        CE == MaxCE.

/*This rules sends the responsibilities across the network the variable relation contains
the location (IP address/port) of the candidate exemplar (CE)*/
responsibility(@CEN, LocalVar, CE, R) :-
        responsibilityEvent(@N, LocalVar, CE, NR),
        variable(@N, CE, CEN).

sentResponsibility(@N, LocalVar, CE, R) :-
        responsibilityEvent(@Node, LocalVar, CE, R).
/*Update availabilities for each iteration*/
/*Availabilitiy updates for each local variable at a periodic time interval*/
aUpdate(@Node, CE, Var) :-
        periodic(@N, E, AP_EPOCH),
        localVariable(@N, CE, _),
        similarity(@N, CE, Var, _).
```

Figure C.1. Overlog for Affinity Propagation (Part I).

```
/* The summation in Equation 4.6*/
sumRp(@N, Var, CE, a_SUM<RP>) :-
        aUpdate(@N, CE, Var),
        rp(@N, OtherVar, CE, RP),
        OtherVar != Var,
        OtherVar != CE.

/*Availability calculations Equation 4.6*/
availabilityCalc(@N, Var, CE, NA) :-
        sumRp(@N, Var, CE, SumRP),
        responsibility(@N, CE, CE, R),
        A := SumRP + R,
        NA := f_min(A, 0.0),
        Var != CE.

/*Self-availability calculations Equation 4.7*/
availabilityCalc(@N, CE, CE, NA) :-
        sumRp(@N, Var, CE, SumRP),
        NA := SumRP,
        Var == CE.

/*This rules sends the availabilities across the network the variable relation contains
the location (IP address/port) of the Var variable*/
availability(@VarNode, Var, CE, NA) :-
        availabilityCalc(@N, Var, CE, NA)
        variable(@Node, Var, VarNode).

/*Exemplar calculation*/
/*Exemplar updates for each local variable at a periodic time interval*/
eUpdate(@N, LocalVar) :-
        periodic(@N, E, AP_EPOCH),
        localVariable(@N, LocalVar, _).

/*Exemplar calculations from Equation 4.8*/
exemplarCalc(@N, LocalVar, a_MAX<Sum>) :-
        eUpdate(@N, LocalVar),
        availability(@N, LocalVar, CE, A),
        sentResponsibility(@N, LocalVar, CE, R),
        Sum := A + R.

exemplar(@N, LocalVar, CE) :-
        exemplarCalc(@N, LocalVar, MaxSum),
        availability(@N, LocalVar, CE, A),
        sentResponsibility(@N, LocalVar, CE, R),
        MaxSum == A + R.
```

Figure C.2. Overlog for Affinity Propagation (Part II).